LotterySampling: A Novel Algorithm for the Heavy Hitters and the Top-k Problems on Data Streams

Gonzalo Solera Pardo

Thesis director: Conrado Martínez

September 2018

Contents

1 Introduction

1.1 Context and stakeholders

The rapid growth of data volume during the last decades has increased the importance of designing efficient data mining algorithms. In particular, the algorithms that operate with massive data streams need to be specially fast and memory efficient to handle the current requirements of the industry.

In this project, new efficient and accurate algorithms are presented that operate on very long streams over a large cardinality set of distinct elements.

This kind of data streams can be found very frequently in network traffic contexts. For example, the IP addresses of the packets sent through a router form a large data stream. This stream is composed by a very large cardinality set of addresses (2^{128}). It's usually interesting to perform a real-time analysis of those IP addresses to detect anomalies, prevent DDoS attacks or simply to analyze the kind of traffic that is currently flowing through the router. On these situations, the information of **which elements (IP addresses) sent through the stream are the most frequent** is essential. In the DDoS example, this information will help to identify who the attackers are since their IP addresses will appear many more times than the rest of addresses.

Another case in which that same question has special relevance is in Internet services that operate at big scale. For example, companies like Google or Amazon tend to create region based caches of the most popular content, so that the latency that their users experience when accessing to it decreases. To achieve it, they perform analysis of the current trends to find popular queries or products. We can then model the sequence of queries or products as a data stream, which also has a very large cardinality of distinct elements (all the possible queries or all the products in the catalog). These companies want to find the frequent elements **efficiently** and **accurately**. However, this is not as trivial as it may sound in such big streams, and a trade-off between the two is needed.

The reasons of why such a "simple" problem is not trivial are related to memory and time constraints. The naïve approach would consist in keeping a counter for each distinct element that is sent through the stream and increasing it for each apparition. However, due to the large cardinalities of the streams, keeping that quantity of counters in memory is unfeasible (because of the memory limitations of the devices) or it's too costly. Also, in these examples time is crucial and we can only allow ourselves to spend a few milliseconds of computation for each element sent through the stream.

As the reader can imagine, there is a big and diverse amount of use cases in which this kind of queries about a data stream are relevant. Apart from the previous examples, domains like telecommunications, Internet advertising, high-

performing databases, stock trading platforms or logging systems are specially interested in data mining techniques were some information can be extracted from the stream, without the need of storing it in memory, which is totally unfeasible. On all those cases, data mining algorithms like the ones we propose have significant interest nowadays.

1.2 Problem formulation

Consider a (long) data stream $\mathcal{Z} = (z_1, \ldots, z_N)$, where each z_i is drawn from some domain or *universe* $\mathcal{U}$ of large cardinality. We will call each z_i a data stream item (or item for abbreviation), and we will refer to the distinct elements of the stream simply as elements. Let $n \leq N$ be the number of elements. We may thus look at the set $X = \{x_1, \ldots, x_n\}$ underlying $\mathcal{Z}$ where x_1, ..., x_n are the n distinct elements that occur in $\mathcal{Z}$.

We will also use $f(x_j)$ to denote the number of occurrences (absolute frequency) of x_j in $\mathcal{Z}$. Furthermore, we can consider that $z_i = x_j$ iff the item z_i is an apparition of the element x_j. Hence, we will use z_i interchangeably with x_j in some situations. For example, if $z_i = x_j$ we may use $f(z_i)$ to refer to $f(x_j)$.

We will assume, w.l.o.g., that we index the elements in X in non-increasing order of frequency, thus $f(x_1) \geq f(x_2) \geq \cdots \geq f(x_{n-1}) \geq f(x_n) > 0$. We will use $p(x_j) = f(x_j)/N$ to denote the relative frequency of x_j. For simplicity, we will assume that $f(x_1) > f(x_2) > \cdots > f(x_n)$ in the definitions below—they can be more or less easily adapted to cope with elements of identical frequency.

The two problems (or type of queries) that we want to study here are:

1. Top-k most frequent elements. Given $\mathcal{Z}$ and a value $k \leq n$, we want to find $\{x_1, \ldots, x_k\}$ (or any subset of k distinct elements with maximal frequencies).

2. Heavy Hitters. Given $\mathcal{Z}$ and a value ϕ, $0 < \phi < 1$, we want to find (or count) the number of distinct elements in $\mathcal{Z}$ with relative frequency $p(x_j) \geq \phi$. Those elements are called *heavy hitters*. Given the data stream and the value ϕ, we want to obtain $\{x_1, \ldots, x_{k^*}\}$, where k^* is the largest value k such that $p(x_k) \geq \phi$. The value k^* is the number of heavy hitters.

Moreover, in both problems, we might want the algorithm to return the frequency $f(x_j)$ of the retrieved elements. None of these two problems can be solved exactly unless we can keep $\Theta(n)$ elements in memory [3, 4]; thus under the tight memory constraints of the data stream model, we must aim at approximate good solutions. Hence, the algorithms that we describe in this document might return elements which are not among the most frequent elements, or that are not heavy hitters, and rather than the frequencies $f(x_j)$ of the returned elements, we will have to content ourselves with estimations $f'(x_j)$ of the real

frequencies.

We will concentrate in algorithms for the top-k most frequent elements. Notice that there can be at most $\lfloor 1/\phi \rfloor$ heavy hitters in a data stream, and thus an algorithm that retrieves the top $k^* = \lfloor 1/\phi \rfloor$ most frequent elements will obtain all the heavy hitters.

In this document, we will also call heavy hitters all the top-k most frequent elements. So we will informally use the term heavy hitter in a more broad sense to mean an element frequent enough that it should be returned by the corresponding algorithm (whether for Top-k or for Heavy Hitters problems).

Normally, the streams that we want to obtain the heavy hitters from are skewed. This means that the frequency of the heavy hitters is notably higher than the non-heavy hitters' frequency (although this frequency might still be very small). We will consider those streams as "easy" and when the skewness is very low, "difficult".

Finally, we want to mention that there exists a more general setup of these two problems, in which every item z_i of the stream has an associated weight w_i. In this general setup, $f(x_j) = \sum_{\{z_i \in Z | z_i = x_j\}} w_i$, which means that the absolute frequency of an element is the sum of the weights of all its apparitions in the stream. As an example, consider a stream formed by a sequence of IP packets from which we want to identify the IP addresses that send most data. We may model the problem using the size of the payload of the IP packets as the weights. The setup we have described previously would be the particular case in which $w_i = 1$, and although our algorithms can be generalized for this more general setup, we will focus in the simpler one.

1.3 State of the art

There exist numerous and wide used algorithms that solve these two problems. For such an algorithm, it's important that:

- It spends as little time as possible for each item of the stream.
- It uses as little memory as possible to keep track of the sample.
- It spends as little time as possible to answer a query.
- It answers the queries with high accuracy.

Normally we find a compromise between these four goals. Some of the algorithms sacrifice memory to get more accurate results, some others spend less time processing an element in exchange of accuracy, etc. However, all of them share some general schema:

They all store a subset of the universe $\mathcal{U}$ in a sample S of size m (which is commonly a parameter of the algorithm). The goal is to keep in the sample S the m most frequent elements seen so far. Apart from the elements themselves, more information is kept and updated to be able to answer the queries, and to be able to update the sample in the future. So, for example, for each element $x \in S$, a counter is kept to store $f'(x)$ which estimates the real frequency $f(x)$. That way, when the algorithms receive a query about the data they have seen so far, they examine the sample and they try to return the most accurate answer, based on the information kept. For instance, for a top-k most frequent query, they might return the k elements with largest estimated frequencies $f'(x)$ (that is the maximum likelihood estimator based on the available information).

The extra information that they store and the way they use it, allows us to distinguish two big groups: Counter-based and sketch-based algorithms.

1.3.1 Counter-based algorithms

This type of algorithms uses an individual counter $f'(x)$ for each element x monitored in the sample and it doesn't keep information of the elements that aren't sampled. For each $z_i \in \mathcal{Z}$, if $z_i \in S$ then its counter is incremented, since z_i was already being sampled and we just need to register its new apparition in the stream. If $z_i \notin S$, then z_i is disregarded or some algorithm-dependent action is taken (which commonly consists in replacing another element from S by z_i).

The new algorithms that we propose in this project fall into this category.

Some of the best known, accurate and efficient counter-based algorithms are SpaceSaving [2], Frequent [3], LossyCounting [6] and StickySampling [6]. They are briefly explained below (for a detailed description, we refer the reader to their original papers).

LossyCounting divides the stream $\mathcal{Z}$ in windows of w elements. It stores the non-sampled elements from window r in S. At the end of window r, it deletes all the monitored elements from S whose counters are less than r. When an element is inserted during window r, it's given the benefit of the doubt and its counter is initialized with $r - 1$. Also the maximum over-estimation of its frequency $(r - 1)$ is recorded for the new item. As a disadvantage of this algorithm, it has to do extra work on windows' boundaries to delete the elements. Also note that the size of the sample varies over time, and it's dependent on the length of the stream, which generally is something that we want to avoid.

StickySampling is a randomized algorithm that also breaks $\mathcal{Z}$ into windows of non-decreasing length w_r, for a window r. The probability that an element gets sampled into S decreases as r (and w_r) increases. At windows' boundaries, for every monitored element, a coin is tossed until a success occurs. Its counter is

decremented for every unsuccessful toss, and the element is deleted if it reaches 0. It also implies extra work on windows' boundaries and a non-fixed-size sample S.

Frequent extends early work done in [7]. When $|S| < m$, S is filled by the next distinct elements that appear in the stream, initializing their counters to 1. When $|S| = m$ and an element $x \in S$ appears in the stream, its counter is incremented. If $x \notin S$ then the counters of all the elements in s are decreased by one. When a counter reaches 0, the counter's element is removed from S and the next element $x \notin S$ that appears in $\mathcal{Z}$ will take its spot and be inserted into S.

SpaceSaving is one of the most popular algorithm (to our knowledge) since it is intuitive, it is very efficient and reasonably simple to implement. It also offers tight and improved error bounds and interesting theoretical guarantees. It keeps a sorted data structure for the sample S, in which the elements are ordered by the values of their counters. When an element $x \in S$ appears in the stream, its counter is updated. If it wasn't in S, the sampled element with lower counter is replaced by x. And x inherits the replaced element's counter (stealing its estimated frequency) and it's incremented by one. Later, we will analyze deeper this algorithm since some of our new ones are based on it. But basically, it's important to note that it has constant time cost to process a new element thanks to how the data structure for S is chosen. Also it can answer the queries efficiently since the data structure is already sorted.

1.3.2 Sketch-based algorithms

This type of algorithms doesn't monitor a subset $S \subset U$. Instead, they monitor all the elements in U in a clever way, using a family F of random hash functions. They work in a similar way as a Bloom Filter works. They have a set of counters which are shared by all the elements from U (like the entire bitmap is shared in a Bloom Filter).

Basically, for each z_i, a subset of $|F|$ counters is chosen using the hash functions from F. Then, those counters are modified. The frequency of an element can be queried by observing the values of its "representative" counters and giving a value depending on them. However, some loss of accuracy is expected due to hashing collisions.

CountSketch [4] finds the representative counters of an element and increments some of them while decreasing the others. The estimated frequency of an element will be the median of its representative counters. To be able to answer the queries for the problems that we deal with in this thesis (Heavy Hitters and Top-k), it's necessary to keep a heap with the m elements with highest estimated frequency, which will be the sample S.

CountMin [5] is a modification of CountSketch. The only difference is that all the representative counters are incremented, and the estimated frequency of an element is the minimum value of its representative counters. This way, the estimated frequency will be an upper bound of the real frequency of that element, since that counter has been incremented in each apparition of it (plus some more times due to collisions of other elements).

2 Metrics

In order to evaluate and compare the performance of the algorithms with the different streams, we define and use some metrics.

The metrics will measure efficiency and accuracy. For the efficiency, the used metrics are the traditional measures such as running time and memory consumption. To evaluate the accuracy/quality of the algorithms we will use conventional metrics, namely, recall and precision, often used in the literature. But in our view these two metrics fail short to capture the main relevant features of the problem and to discriminate well among the different algorithms, and thus we have introduced a new metric which we discuss in this section in detail.

2.1 Recall and precision

As we have mentioned, we will use the standard definitions of recall and precision from information theory. These two metrics where developed and are extensively used in the area of Information Retrieval, including in the existing literature for the Heavy Hitters and Top-k problems. The formal definition is detailed bellow.

Let A be the set of heavy hitters, and let B be the set of elements returned by the algorithm when answering a query. Let $\text{rank}(x)$ be the position of x in S sorting by $f'(x)$.

Hence, for the Top-k problem:

$$A = \{x_j \in \mathcal{U} | 1 \leq j \leq k\} \qquad B = \{x_j \in S | \text{rank}(x_j) \leq k\}$$

And for the Heavy Hitters problem:

$$A = \{x_j \in \mathcal{U} | f(x_j) \geq \phi\} \qquad B = \{x_j \in S | f'(x_j) \geq \phi\}$$

Then, we define recall as:

$$R = \frac{\sum_{x \in A \cap B} 1}{\sum_{x \in A} 1} = \frac{|A \cap B|}{|A|}$$

And precision as:

$$P = \frac{\sum_{x \in A \cap B} 1}{\sum_{x \in B} 1} = \frac{|A \cap B|}{|B|}$$

Note that $0 \leq R \leq 1$ and $0 \leq P \leq 1$. Also note that $R = 1 \iff A \subseteq B$, meaning that the recall is maximum iff all the heavy hitters are returned by the algorithm. $R = 0$ iff no heavy hitter is returned. And similarly for the precision, $P = 1 \iff B \subseteq A$ meaning that all the elements returned are heavy hitters, and $P = 0$ iff none of the returned elements is a heavy hitter (assuming that $B \neq \emptyset$).

An intuitive understanding of the recall is that it's a metric that requires the algorithms to return as many heavy hitters as possible. And the precision requires that the returned elements are as "pure" as possible since it measures the proportion of heavy hitters among the returned elements. So an algorithm that returns all the heavy hitters by returning a huge set of elements, it will have a high recall but low precision. And vice versa, if the algorithm returns a small set of heavy hitters but only heavy hitters, the precision will be high but the recall will be low. So the goal of an algorithm is to return exactly the heavy hitters, that is to be as close as possible to the "ideal" $R = P = 1$.

On the other hand, notice that $R = 0$ if and only if $P = 0$.

Last but not least, in the case of the Top-k problem (which is the one in which we have focused our efforts), we have $|A| = |B| = k$, that is, for the Top-k problem recall and precision are identical ($R = P$).

2.2 Weighted recall

We propose a generalization of the recall that we call weighted recall, or R_w. To our knowledge, this metric is new and we haven't seen it in the existing literature.

The idea is to have a metric that penalizes more to miss an element x_a than an element x_b with $a < b$. So we weight the "importance" of an element x by using its relative frequency $p(x)$. Hence:

$$R_w = \frac{\sum_{x \in A \cap B} p(x)}{\sum_{x \in A} p(x)}$$

Note how we also have that $0 \leq R_w \leq 1$.

We believe that this metric is very useful since in most of the use cases, the harmful effect of missing the top-1 heavy hitter is much bigger than the effect of missing the k^{th} heavy hitter.

However, we can't use a similar generalization of the precision, since the straightforward definition of P_w would reward an algorithm that retrieves less

frequent elements. The definition of weighted precision would be:

$$P_w = \frac{\sum_{x \in A \cap B} p(x)}{\sum_{x \in B} p(x)}$$

And a situation in which P_w rewards an algorithm that returns worse elements would be, for example, the following: Consider B and B' such that $A \cap B = A \cap B'$ and $|B| = |B'|$. If $\forall_{x \in B} \forall_{x' \in B'} f(x) \geq f(x')$, then $P_w(B) \leq P_w(B')$. This happens because the two expressions share the numerator, and the one with lower denominator (B') would obtain a higher precision. But if B' has a lower denominator, then B' contains less frequent elements than B.

Hence, for the Top-k problem we will use the precision (which is the same as the recall) and the weighted recall during the experiments.

2.3 Squared error

We will also use the well known squared error of the estimated frequencies. More formally:

$$E = \sum_{x \in \mathcal{U}} (f(x) - f'(x))^2,$$

where by convention, we take $f'(x) = 0$ if $x \notin B$.

$E \geq 0$ will be minimum when $A = B$ and $f'(x) = f(x)$ for $x \in B$.

3 The algorithms

The main algorithm that we propose in this thesis is called LotterySampling. However, we will describe other simpler versions first, constructing LotterySampling iteratively.

This thesis has had a big component of exploration and most of the efforts have been applied to discover the algorithms and to experiment and improve them iteratevely. However, the depth of the theoretical analysis is limited, since we considered it out of scope. We will focus on giving the intuitions of why they work well and the ideas on which they are based. We will leave for future work a more in-depth analysis and formal proofs. For example, the ultimate goal would be to include LotterySampling into the δ-ϵ-deficient framework in which other state-of-the-art algorithms are in.

3.1 BasicLotterySampling

All of our algorithms are based on the idea of what we call lottery tickets and lottery tokens. A lottery token (or token) is an independent uniform random number between 0 and 1. Basically, we will generate one of such tokens for each

item z_i that appears on the stream. And each element x_j will have its own lottery ticket (or ticket), which will be the highest token obtained among all its apparitions so far.

In BasicLotterySampling we will have a sample S of fixed size m. And the elements that will be in this sample are the ones with highest lottery ticket. The reason behind using the word "lottery" is that we can define an analogy with a regular lottery: In a regular lottery, there are some winners between the participants, and they are randomly chosen by selecting a subset of lottery tickets. The more lottery tickets a participant has, the more chances it will have of winning. And similarly in BasicLotterySampling, the more frequent an element is, the more lottery tokens it will get, so it will have more chances of getting a high lottery ticket. If its lottery ticket is among the m-highest lottery tickets, it will stay in the sample, or following the analogy, be one of the winners of the lottery.

We can formalize more the previous explanation:

Let $t(z_i)$ be the token obtained by the i^{th} item. And let $t^*(x_j)$ be the ticket of x_j at any given time. Then BasicLotterySampling is described in Algorithm 1:

Algorithm 1: BasicLotterySampling(m)

```
S = ∅
for i = 1 to N do
    x = element(z_i)
    if x ∉ S then
        if |S| < m then
            S.insert(x)
            t*(x) = t(z_i)
            f'(x) = 1
        else
            x* = argmin_{x'∈S} t*(x')
            if t(z_i) > t*(x*) then
                S.remove(x*)
                S.insert(x)
                t*(x) = t(z_i)
                f'(x) = 1
    else
        f'(x) = f'(x) + 1
        t*(x) = max(t*(x), t(z_i))
```

To answer a k-top query, the k elements from S with highest $f'(x)$ are returned. Note that $f'(x)$ won't affect which elements are in S, it will only determine which elements from the ones in S are returned to answer a query,

and in which order.

Note that we initialize $f'(x) = 1$ when we insert a new element x in the sample while replacing another. However there are other alternatives. Let $f_{obs}(x)$ be the observed frequency of x (i.e., the count of apparitions of x from when it entered into S), and let $f_{init}(x)$ be the initial frequency that we estimate when x enters into the sample. Then, $f'(x) = f_{init}(x) + f_{obs}(x)$. In Algorithm 1 we used $f_{init} = 0$, so $f(x) \geq f'(x) = f_{obs}(x)$. However, we propose two other estimations of the initial frequency:

- $f_{init}(x_j) = \mathbb{E}[f(x_j)|T^{(j)} = t^*(x_j)] = \left\lfloor \frac{t^*(x_j)}{1-t^*(x_j)} \right\rfloor$. That is, we estimate the initial frequency of x_j by calculating the frequency it needed to get its lottery ticket if $\mathrm{Var}[T^{(j)}]$ were 0. The definition of the random variable $T^{(j)}$ is detailed in the next chapter.
- $f_{init}(x) = \left\lfloor \frac{1}{1-t^*(x^*)} \right\rfloor$. The probability for x_j to enter the sample in this apparition was $1 - t^*(x^*)$. And the idea is to assume that this probability has always been the same, so we can model it with a geometric distribution. So this is an upper bound of the expectation of the number of times x needed to appear until it entered into the sample.

The estimation that seems to work better in our experiments is $f_{init}(x) = 0$. Hence, we will use this estimation in the experiments.

3.1.1 Analysis

Let $t = t^*(x^*)$. We can view t as a "threshold", since it's the value that the items need to beat with their tokens to enter the sample. Note that the threshold never decreases and it always increases after an insertion (when $|S| = m$).

Let $T_r^{(j)} = t(z_i)$ such that z_i is the r^{th} apparition of x_j in the stream $\mathcal{Z}$. Or equivalently, the token obtained by element x_j on its r^{th} apparition. Observe that $T_r^{(j)}$ are independent uniformly distributed random variables between 0 and 1. Hence:

$$\Pr\left[T_r^{(j)} \leq y\right] = y$$

Let $T^{(j)} = t^*(x_j)$. $T^{(j)}$ is also a random variable and can be expressed in terms of $T_r^{(j)}$ as follows:

$$T^{(j)} = \max\left(T_1^{(j)}, T_2^{(j)}, \ldots, T_{f(x_j)}^{(j)}\right)$$

The probability distribution of $T^{(j)}$ is:

$$\begin{aligned}\Pr\left[T^{(j)} \leq y\right] &= \Pr\left[T_1^{(j)} \leq y \wedge T_2^{(j)} \leq y \wedge \ldots \wedge T_{f(x_j)}^{(j)} \leq y\right] \\ &= \prod_{r=1}^{f(x_j)} \Pr\left[T_r^{(j)} \leq y\right] \\ &= y^{f(x_j)},\end{aligned}$$

which is the cumulative distribution function (CDF). Taking derivatives with respect to y, we obtain the probability density function (PDF) of $T^{(j)}$:

$$f(x_j) \cdot y^{f(x_j)-1}$$

The expectation of $T^{(j)}$ is then:

$$\begin{aligned}\mathbb{E}\left[T^{(j)}\right] &= \int_0^1 y \cdot f(x_j) \cdot y^{f(x_j)-1} dy \\ &= \int_0^1 f(x_j) \cdot y^{f(x_j)} dy \\ &= \frac{f(x_j)}{f(x_j)+1}\end{aligned}$$

Note that, for any given x_a and x_b, $\mathbb{E}[T^{(a)}] > \mathbb{E}[T^{(b)}]$ if and only if $f(x_a) > f(x_b)$. And then, $\mathbb{E}[T^{(1)}] > \mathbb{E}[T^{(2)}] > \ldots > \mathbb{E}[T^{(n)}]$. And this is the powerful idea that we try to exploit in our algorithms. We don't know $f(x_j)$ but we can easily keep track of the k highest $T^{(j)}$ as detailed before, and in theory we would be capturing exactly the top-k heavy hitters. A visual explanation of this can be found in Figure 1 with $m = 3$ and $n = 5$.

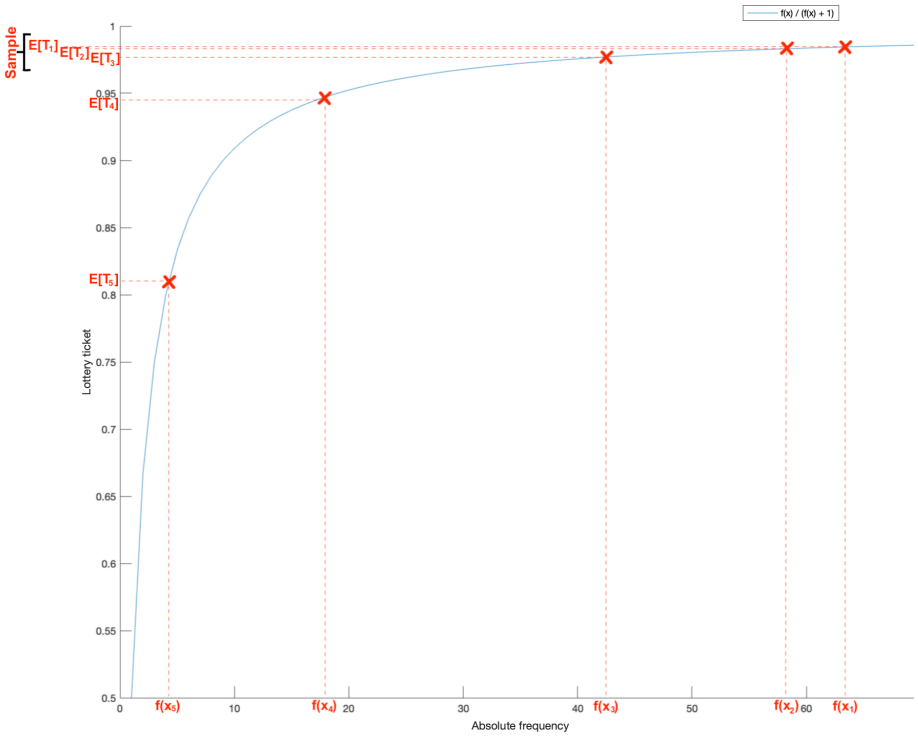


Figure 1: Illustration of BasicLotterySampling

So if $T^{(j)} = \mathbb{E}[T^{(j)}]$ then we could solve the problem exactly. Obviously this is not the case. Because of the variance $\text{Var}[T^{(j)}]$, the random variables $T^{(j)}$ won't always have the value of their expectation. i.e., One element may obtain a higher (or lower) ticket than the one it "deserves" according to its frequency. This may result in keeping in S a less frequent element instead of another with higher frequency. For example, in the example from Figure 1, if $t^*(x_4) > t^*(x_3)$, then x_4 will be wrongly kept in the sample instead of x_3. In fact, the probability of this happening is $\frac{f(x_4)}{f(x_3)+f(x_4)}$, because:

$$\begin{aligned}
\Pr\left[T^{(a)} > T^{(b)}\right] &= 1 - \Pr\left[T^{(a)} \leq T^{(b)}\right] \\
&= 1 - \int_0^1 \Pr\left[T^{(a)} \leq y | T^{(b)} = y\right] \cdot \Pr\left[T^{(b)} = y\right] dy \\
&= 1 - \int_0^1 y^{f(x_a)} \cdot f(x_b) \cdot y^{f(x_b)-1} dy \\
&= \frac{f(x_a)}{f(x_a) + f(x_b)}
\end{aligned}$$

The rest of the algorithms that we present in this work try to decrease the

harmful effect of the variance.

There is also another challenge that all of our algorithms will face. Note that:

$$\lim_{N\to\infty} \mathbb{E}[T^{(j)}] = 1 \qquad\qquad \lim_{N\to\infty} \mathrm{Var}[T^{(j)}] = 0$$

This is due to the fact that while maintaining the distribution of the stream $\mathcal{Z}$, the theoretical relative frequencies $p(x_j)$ will stay the same, but the absolute frequencies $f(x_j) = p(x_j) \cdot N$ will increase, having more tokens, which makes the expectation tend to 1 and the variance tend to 0. So although we can expect lottery tickets nearer to their expectation, their values will be very close between them and, although the variance will be small, a very little variance will affect enormously which elements will be kept in the sample.

A different way to illustrate the problem of the variance is the following: There are k heavy hitters in the stream and $n-k$ non-heavy hitters. The sum of frequencies of the non-heavy hitters is $Q = N - (f(x_1) + f(x_2) + \ldots + f(x_k))$. Consider k groups of the non-heavy hitters such that the sum of apparitions of the elements in each group is approximately $W \approx \frac{Q}{k}$. In difficult streams (where the skewness is not very big), $W \gg f(x_1)$, and specially $W \gg f(x_k)$. This means that each of the k groups will collect W tokens among its members and we expect at least one non-heavy hitter on each group to get a token (and thus a ticket) higher than the ticket of some heavy hitter(s). This will result in replacing the heavy hitters by non-heavy hitters in S. We name this phenomenon "element alliance", since a way to interpret it is that non-heavy hitters will combine their tokens to obtain very high tickets such that one of them will enter S.

BasicLotterySampling doesn't work well because of these reasons, as shown later in the experiments.

3.1.2 Asymptotic cost

The memory cost of BasicLotterySampling is $O(m)$ since we just keep a counter and a ticket for every element in the sample, whose size is upper bounded by m.

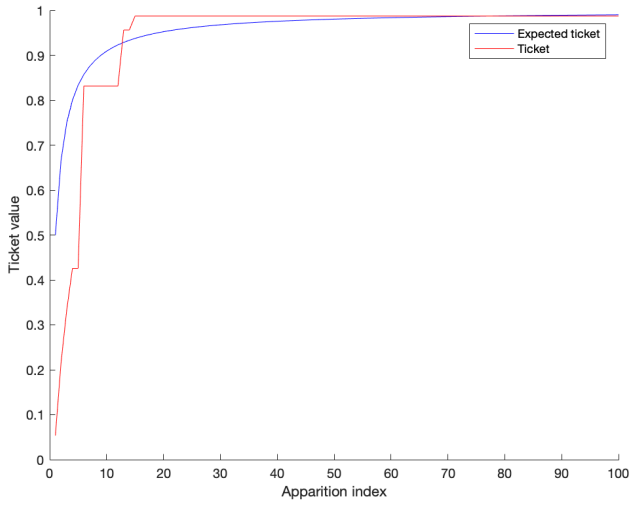
The time cost per each item is $O(\log(m))$ since we have to store S in a min heap in order to retrieve the element with minimum ticket fast. Note that for each apparition z_i of the element $x_j \notin S$, we will only insert it with probability $1-t$, where t is the threshold. As we process the stream, the current threshold never decreases—it actually increases whenever an element enters the sample and it might increase whenever we process an element already in the sample—and hence, more and more items will be ignored. Processing each of these items has cost $O(1)$.

Note that if we want to answer queries efficiently, we should also sort S by $f'(x)$ to answer the top-k queries in $O(k)$ time, which doesn't increase the asymptotic cost. Finally, note that we need a hash map to efficiently decide membership of S and to locate the elements in S.

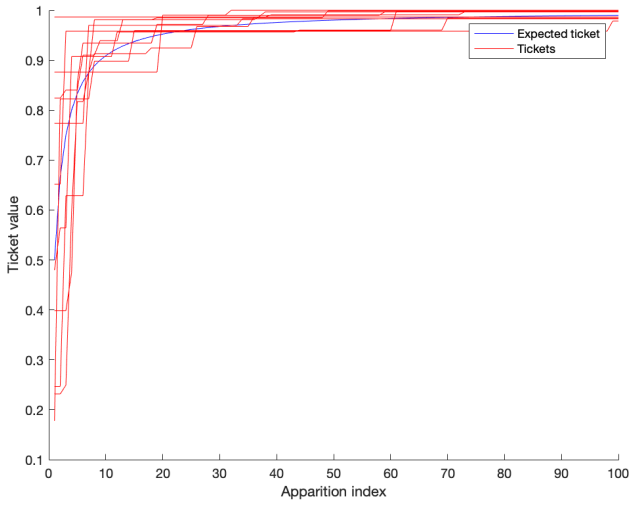
3.2 ParallelLotterySampling

One approach that we propose to solve the problem of the variance that BasicLotterySampling has is to independently run H parallel instances of BasicLotterySampling with different random seeds. The idea is to combine the query answers of the H instances into one, decreasing the harmful effect of the variance. We call this algorithm ParallelLotterySampling.

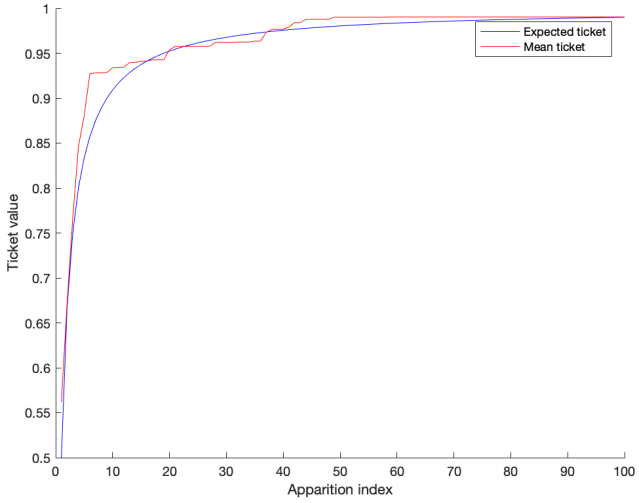
Before detailing ParallelLotterySampling, the motivation is shown empirically through different experiments in figure 2. What we aim to show is that the variance of the average of H tickets (H maximum tokens from H different executions) decreases when we increase H. The Subfigure 2a shows the evolution of the lottery ticket $T^{(j)}$ for a given element x_j through a stream in a single execution. And we also plot the expected lottery ticket for each frequency. So for every apparition of x_j (x axis), $T^{(j)}$ and $\mathbb{E}[T^{(j)}]$ (y axis) are updated. In Subfigure 2b we also plot $T^{(j)}$ but for 10 different executions. In Subfigure 2c we plot the evolution of the average of the $T^{(j)}$s from the 10 different executions. And in Figure 2d we plot the same but for 100 executions.



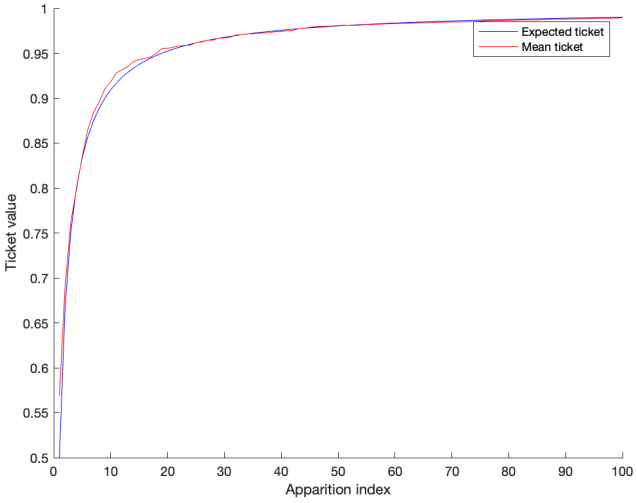
(a) Evolution of ticket from one execution



(b) Evolution of tickets from 10 executions



(c) Evolution of mean ticket from 10 executions



(d) Evolution of mean ticket from 100 executions

Figure 2

The problem that BasicLotterySampling had was the variance of $T^{(j)}$. And as shown in Figure 2, by applying the average of H tickets from H independent executions, the averaged ticket is much closer to the expectation. In fact, let $T'^{(j)}$ be the averaged ticket of x_j from H independent executions. Then $\mathrm{Var}[T'^{(j)}] = \frac{\mathrm{Var}[T^{(j)}]}{H}$.

At first glance this seems to solve the problem with a big enough H. However, this implies knowing the H tickets of each distinct element. i.e, An element might not have a top-m ticket in all the H executions, and we still need those tickets to do the average.

So instead of trying to decrease the variance of $T^{(j)}$, we will try to decrease the variance of BasicLotterySampling by executing it H times and by combining the results afterwards. So the intuition is: If an element is one of the heavy hitters, then we expect it to get a large enough ticket in some of the H instances to get inside the corresponding samples. Then we use some strategy to unify the results from the H instances.

More formally, ParallelLotterySampling runs H instances of BasicLotterySampling. That means that for each item z_i, it will generate H tokens and update the samples S_h (where $1 \leq h \leq H$) independently following the description in Algorithm 1. Let S be the union of all the S_h. We can view S as a sample that contains the elements that obtained a high ticket in some of the S_h samples. We may consider ParallelLotterySampling as a Monte Carlo algorithm: the probability for a given heavy hitter to appear in S will increase with the number of instances. The heavy hitters will have more preference to appear in the answer of a query thanks to the strategy that unifies the results from the H samples.

Note that (after the initial phase in which the samples S_h get filled):

$$m = |S_h| \leq |S| \leq m \cdot H$$

Also note that there will constantly be $m \cdot H$ tickets, independently on the number of distinct elements in S.

If we use $f_{init}(x) = 0$ for each one of the H instances, then it would make sense to share the counter $f'(x)$ between all the instances (having $|S|$ many counters). This means that for each $x \in S$ we only have a single counter to keep $f'(x)$, instead of one counter for each sample in which x is. This single counter counts all the apparitions of x since it was inserted in the first sample S_h. Consider for example that a certain element x not yet sampled any sample, enters sample S_3. Then it appears three more times and then, on fifth occurrence (since it entered S_3) it also enters S_1. It appears two more times, but then it's evicted from S_3. Afterwards, it appears three more times while staying in S_1. Then $f'(x) = 10$ at that moment, instead of having different counters for S_1 and S_3 with values 6 and 7 respectively (the counter from S_3 would be deleted when x is evicted from it). If we use another initialization $f_{init}(x)$ then each instance of BasicLotterySampling that contains x in its sample may keep a separate counter $f'(x)$ for x, so in such a case no counter would be shared between samples.

To answer a top-k query, ParallelLotterySampling will use a strategy to determine the k most frequent elements using the information from the H samples. There is a big amount of possible strategies, and they depend on how $f'(x)$ is defined. Some of them are described below:

- If we keep a single counter $f'(x)$ for each x that appears in at least one of the H samples (this is possible when we have $f_{init}(x) = 0$) then we propose the following two alternatives:
  - The k elements with higher $f'(x)$ are returned to answer a query. It is a simple strategy and we only use the observed frequencies of the elements in S.
  - For each element x in S, we compute the average of its tickets from all the samples. If x is not inside some sample S_h then we use as ticket $\frac{f'(x)}{f'(x)+1}$ which is a lower bound of the ticket it deserved, $\frac{f(x)}{f(x)+1}$. The k elements with higher averaged ticket are returned.
- If for an element x we use a different counter $f'(x)$ for each sample (so no counter is shared between instances, which is not possible when $f_{init}(x) \neq 0$), then we propose two other alternatives:
  - For each x in S, we find the maximum $f'(x)$ among all the samples where x appears. The k elements with higher maximum estimated frequency are returned.
  - Similar as before, but instead of the maximum we compute the average of $f'(x)$ from the H samples (taking $f'(x) = 0$ whenever x does not belong to the corresponding sample). The k elements with highest average estimated frequency are returned.

The strategy that seems to work better in our experiments is the first one: using $f_{init}(x) = 0$ and sharing the $f'(x)$ between samples, sorting by $f'(x)$ to answer the queries. Hence, we will use this strategy in the comparisons against other algorithms. The pseudo-code for ParallelLotterySampling can be found in Algorithm 2.

Algorithm 2: ParallelLotterySampling(m, H)

```
S = ∅
S_1 = ∅, S_2 = ∅, ..., S_H = ∅
for i = 1 to N do
    x = element(z_i)
    if x ∉ S then
        f'(x) = 0
    for h = 1 to H do
        if x ∉ S_h then
            if |S_h| < m then
                S_h.insert(x)
                t*_h(x) = t_h(z_i)
            else
                x* = argmin_{x'∈S_h} t*_h(x')
                if t_h(z_i) > t*_h(x*) then
                    S_h.remove(x*)
                    S_h.insert(x)
                    t*_h(x) = t_h(z_i)
        else
            t*_h(x) = max(t*_h(x), t_h(z_i))
    S = ⋃_{S_h} S_h
    if x ∈ S then
        f'(x) = f'(x) + 1
```

3.2.1 Asymptotic cost

The time cost and memory usage of ParallelLotterySampling is much higher. The memory usage is $O(m \cdot H)$ and the time cost to process each item is $O(\log(m) \cdot H)$. Note that each element in S needs to have a hash map to find the samples in which it is inserted, apart from the hash map to locate the elements. We also need to maintain sorted the $|S|$ elements in order to answer queries in $O(k)$ time.

ParallelLotterySampling has similar costs than sketch-based algorithms but it isn't a practical algorithm. It's useful for experimentation in order to measure the performance of other algorithms.

3.3 SpaceSavingThreshold

We aim to use a fusion of BasicLotterySampling with SpaceSaving to build LotterySampling. But first, we will define a very simple probabilistic version of SpaceSaving that we call SpaceSavingThreshold. We will use the same concept

and notation from the previous algorithms of $t(z_i)$, $f'(x)$, $f_{init}(x)$ and $f_{obs}(x)$.

In the original version of SpaceSaving, the sample S is filled by the first m distinct elements using $f_{init}(x) = 0$ and $f_{obs}(x) = 1$ on their insertions. For each apparition of x_j when $x_j \in S$, we increment $f_{obs}(x_j)$ by one. When $x_j \notin S$ and $|S| = m$, then we always insert x_j by replacing the element x^* with lower $f'(x^*)$. In such cases we use $f_{init}(x_j) = f'(x^*)$ and $f_{obs}(x_j) = 1$. This can be interpreted as x_j stealing the estimated frequency of x^*. More formally:

Algorithm 3: SpaceSaving(m)

```
S = ∅
for i = 1 to N do
    x = element(z_i)
    if x ∉ S then
        if |S| < m then
            S.insert(x)
            f'(x) = 1
        else
            x* = argmin_{x'∈S} f'(x')
            S.remove(x*)
            S.insert(x)
            f'(x) = f'(x*) + 1
    else
        f'(x) = f'(x) + 1
```

To answer a top-k query, the k elements with higher $f'(x)$ are returned.

In order to find x^* and to answer the queries efficiently, a data structure for S is suggested in the original paper of SpaceSaving such that updates and insertions are executed in $O(1)$, with $O(k)$ to answer the queries, which is optimal.

This data structure consists in a sorted list of buckets, where a bucket is a list of all the elements with same $f'(x)$. This list of buckets is sorted by $f'(x)$. So to replace the element x^* with lower $f'(x^*)$ from S, we can access the bucket from one end of the list to find it. Note that $f'(x)$ only increases by 1 each time, so in updates and insertions of the element x, we just need to check if the next bucket (in case it exists) contains elements with estimated frequency $f'(x) + 1$. If that is the case, we just have to move x from its current bucket to the next one. If such bucket doesn't exist, we just have to create it next to the current bucket and move x inside. If the previous bucket becomes empty, we delete it. All these operations can be done in $O(1)$ time. Note that we also need a hash map to find the elements in S.

In SpaceSavingThreshold we have a new parameter $0 \leq t < 1$ called "threshold". The only modification respect to SpaceSaving is that we will only insert

an element x with probability $1 - t$ (when $|S| = m$). With this modification, many properties and deterministic guarantees from SpaceSaving are lost, as for example $\sum_{x \in S} f'(x) \neq N$. However, with a properly tuned value of t dependent on the stream, a huge improvement can be achieved in accuracy respect to SpaceSaving. Regarding performance there is also an improvement since there will be many less insertions ($1 - t$ times on average), although the asymptotic cost doesn't change.

SpaceSavingThreshold is detailed in Algorithm 4. Note that we use the same concept of tokens $t(z_i)$ from BasicLotterySampling:

Algorithm 4: SpaceSavingThreshold(m, t)

```
S = ∅
for i = 1 to N do
    x = element(z_i)
    if x ∉ S then
        if |S| < m then
            S.insert(x)
            f'(x) = 1
        else if t(z_i) > t then
            x* = argmin_{x' ∈ S} f'(x')
            S.remove(x*)
            S.insert(x)
            f'(x) = f'(x*) + 1
    else
        f'(x) = f'(x) + 1
```

To understand the improvement respect to the original version it's important to understand how the original version behaves:

When a non-sampled element x appears in the stream, SpaceSaving gives it the benefit of the doubt and considers it a heavy hitter inserting it into S, by expelling another element x^*. Maybe there were more elements with estimated frequency $f'(x^*)$, or maybe there was only x^*. In any case, x will inherit the estimated frequency $f'(x^*)$ and increment it by one, so it will be very close to the "frontier" of S. We call frontier the bucket with smallest estimated frequency. Hence x will be dangerously near to being replaced by another element from the stream. If x receives more hits soon (there are more apparitions of x), it may be able to move away from the frontier (that is, its counter $f'(x)$ becomes significantly larger than the minimum counter $f'(x^*)$ that defines the frontier). If x is actually a heavy hitter, this separation will be easier since it will receive more hits than the non-heavy hitters. Hence, we can intuitively visualize this as a race between the elements trying to "survive" in S, where every time an element enters into S, it starts next to the frontier. The more frequent an element is, the faster it will run away from the frontier. At the same time, the frontier

will move forward every time all the elements in the frontier are replaced or receive a hit.

If there is not much skewness in the stream (i.e., many heavy hitters have frequencies not that different from many non-heavy hitters) most of the elements in S will be very near to the frontier, being replaced constantly without having enough time to run away from it. However, if the heavy hitters have a high enough frequency we will find the heavy hitters sorted by $f(x)$ in buckets far away from the frontier. In the frontier there will be some elements (at least one) that are constantly being replaced. In that case, the sum of frequencies Q of the non-heavy hitters is not high enough for the frontier to move faster and to reach the heavy hitters that have separated from it.

Let $1 \leq V \leq m$ be the number of elements in the frontier, then the frontier will move to the next estimated frequency once every V new insertions (roughly, since the frontier can also move if all the elements in the frontier receive a hit). This means that an element x needs to have a relative frequency $p(x) > \frac{1}{V}$ in order to separate from the frontier (necessary but not sufficient). Then, if S is full of non-heavy hitters (most of them in the frontier), a heavy hitter will have more time to get away from the frontier since the frontier will move slower. However, note that as more heavy hitters get away from the frontier, the frontier will become smaller in size (with less elements), which will increase its "speed" [1].

By introducing the threshold in SpaceSavingThreshold we decrease the speed of the frontier, since less elements will be inserted. Hence, the heavy hitters will have much more time to get away from the frontier, which actually means that they can have a much smaller frequency and still be sampled. So in difficult streams where the skewness is low (and the heavy hitters have a small frequency), a properly tuned value of t will allow us to keep the heavy hitters in S. In fact, an element x needs to have a relative frequency $p(x) > \frac{1-t}{V}$ in order to separate from the frontier (necessary but not sufficient). Note that with values of t close to 1 this requirement is much lower (and better) than in the original version of SpaceSaving, in which $p(x)$ needs to be greater than $1/m$. So we are interested in choosing a high value of t.

However, if we choose a value for t that is too high, then the heavy hitters may not be able to even enter the sample. Hence, we are also interested in having a low enough value of t that guarantees that the heavy hitters will pass the threshold at some point. So for a heavy hitter x we want to have $f(x) > \left\lceil \frac{1}{1-t} \right\rceil$.

The optimal value of t is difficult to define, and it depends on the stream, including the distribution it follows and its length. This makes SpaceSavingTh-

[1] We can formally define the speed of the frontier in the interval $[i, i+D]$ taking the difference between the minimum frequency in the sample when inspecting z_i and the minimum frequency in the sample when inspecting z_{i+D}, and dividing by D.

reshold much less interesting, since it needs tuning and/or previous knowledge of the stream.

3.3.1 Asymptotic cost

As mentioned before, the time cost of SpaceSavingThreshold (and SpaceSaving) is $O(1)$ for both insertions and updates. To answer a top-k query it needs $O(k)$.

The memory cost is $O(1)$. The number of buckets will always be between 1 and m. The union of elements from all the buckets will be all the elements in S.

3.4 LotterySampling

LotterySampling is the main algorithm that we propose in this thesis. It's based on the intuitions explained in the previous algorithms and it's very accurate, substantially outperforming existing algorithms.

It's basically a fusion of SpaceSavingThreshold and BasicLotterySampling. It uses the ideas behind BasicLotterySampling to dynamically choose the value t of SpaceSavingThreshold. It's detailed in Algorithm 5.

Algorithm 5: LotterySampling(m)

```
S = ∅
for i = 1 to N do
    x = element(z_i)
    if x ∉ S then
        if |S| < m then
            S.insert(x)
            f'(x) = 1
            t*(x) = t(z_i)
        else
            t = min_{x'∈S} t*(x')
            if t(z_i) > t then
                x* = argmin_{x'∈S} f'(x')
                S.remove(x*)
                S.insert(x)
                f'(x) = f'(x*) + 1
                t*(x) = t(z_i)
    else
        f'(x) = f'(x) + 1
        t*(x) = max(t*(x), t(z_i))
```

If x is not in S, we insert it only if its token is higher than the minimum ticket in the sample. However, instead of replacing the element with the min-

imum ticket (as it's done in BasicLotterySampling), we replace the element x^* with lowest estimated frequency $f'(x^*)$ (in case of tie, any is chosen). For that reason what we call ticket in LotterySampling is not exactly a ticket in the sense given in BasicLotterySampling. Because an element x_j could have obtained its maximum token (its ticket) and be in S, be expelled because it had the lowest $f'(x_j)$ at some point and be inserted again with a lower token, so its true ticket gets lost. Hence, in the context of LotterySampling, an element's ticket is its highest token obtained since its last insertion into S. We will use "real ticket" to refer to the original definition of ticket.

Note how in LotterySampling, the threshold t never decreases, but it doesn't always increase in insertions (as opposed to BasicLotterySampling, that always increases in insertions). In fact, the threshold only increases if the replaced element from an insertion was the element with minimum ticket, or if the element with minimum ticket receives a hit that increments its ticket. Also note that with the same stream and with the same sequence of tokens, the threshold in LotterySampling will always be equal or lower than the threshold in BasicLotterySampling.

LotterySampling basically follows the idea of BasicLotterySampling, because it intends to keep in S the elements with highest real tickets, but it doesn't relay completely in the tickets. So if a non-heavy hitter gets very lucky and obtains a very high ticket when entering into S, it won't mean it will stay in the sample indefinitely, because if it doesn't get enough hits soon it will be expelled from S, since it won't be able to get away from the frontier faster than the real heavy hitters that are in S.

Hence, for an element to be in S, it needs to obtain a high ticket to enter S and to have a high frequency relative to the rest of the elements to avoid being expelled from it.

LotterySampling has the good property from SpaceSavingThreshold that allows the heavy hitters to have a small relative frequency and still be kept in S. Since the threshold t can only increase, the heavy hitters will have increasingly more time to move away from the frontier.

Also, since the non-heavy hitters are not going to survive in S, t will be some heavy hitter's ticket most of the time. That means that t won't increase too fast because the "element alliance" phenomenon suffered by BasicLotterySampling won't affect it that much. So t will have a similar value to the expected real ticket from some heavy hitter x_j with j near to k. Then j out of the k heavy hitters should be able to overpass the threshold (since their expected real ticket is higher), solving the problem of SpaceSavingThreshold when choosing a fixed and (possibly too) high t.

3.4.1 Asymptotic cost

The asymptotic cost per item of LotterySampling is the same as in BasicLotterySampling, since we need to find the lowest ticket in S by using a heap. LotterySampling can use the efficient data structure from SpaceSaving to keep the elements sorted by $f'(x)$ with cost $O(1)$, although it will be dominated by the cost of the heap.

Then the time cost of LotterySampling is $O(\log(m))$ and it has a memory usage of $O(m)$. To answer a top-k query it will take $O(k)$ by using the SpaceSaving's data structure.

3.5 Other variations

In the exploratory phase of this thesis we considered some ideas to improve the algorithms or to solve some of their problems. However, we haven't fully experimented with them so they are not merged in the proposed version of LotterySampling nor are they included in the experiments of this document. We summarize some of these ideas in the following subsections.

3.5.1 LotterySampling with ticket scaling

One implementation problem that LotterySampling has is the resolution of the tokens and tickets. The threshold will approach arbitrarily to 1 with arbitrary long streams, so the precision of the binary representation will affect the algorithm. Consider a natural number $v \in [0, 2^b)$ using b bits to represent a ticket $t^*(x)$, such that $t^*(x) = v/2^b$.

One possible solution that we call "ticket scaling" consists in scaling the tickets when the threshold overpasses the value $1 - 2^{-r}$ for any r. Note that in such cases, the threshold will contain r leading ones in its binary representation v. Since the threshold never decreases and all the other tickets in S need to be equal or higher, then all the current tickets in S will also have r leading ones in their binary representations. So we can always remove as many leading ones that the threshold has from all the tickets in S, and keep track of the number of removed leading ones so far. The removal of the ones is done by shifting the binary representation of the tickets to the left and adding zeros to the right. This way, we regain precision by not wasting bits that will always contain ones. The generation of the next tokens needs to be slightly adapted to this modification. For example, first a random number is generated to make sure that the new token also has r leading ones, and then we generate the rest of the ticket.

3.5.2 BasicLotterySampling with ticket aging

It is possible to add a degradation or decay with time to the tickets in S, meaning that the tickets will loose value while they get older. Hence, in this variation

the tickets in S will decrease over time following some criteria. This decay could be exponential or linear. The idea is to force the elements to keep getting high tokens, so it wouldn't be enough to get a large one once to stay in the sample.

The problem it has is that it's difficult to choose an aging function for the tickets, and that this function probably needs some extra parameters. We have tested some approaches with both linear and exponential aging and it allows the algorithms to improve substantially but requiring stream-dependent tuning. This idea can also be applied to LotterySampling.

3.5.3 LotterySampling adapted for the weighted setups

As mentioned previously, there exists a weighted version of the Heavy Hitters and Top-k problems in which each item z_i has some weight $w_i \in \mathbb{N}$, and for the element x_j the frequency $f(x_j)$ is the sum of the weights from its apparitions.

We can generalize our algorithms by generating w_i tokens for each apparition z_i of the element x_j, and using the highest one to update S. This would be equivalent to w_i consecutive apparitions of x_j. Then we increment $f'(x_j)$ by w_i instead of by 1.

The problem with this approach is that the cost per item would be $O(w_i + \log(m))$. To solve it, we can compute one single token as $t'(z_i) = \sqrt[w_i]{t(z_i)}$. The random variable $t'(z_i)$ is distributed as the maximum of w_i independent uniforms in $(0, 1)$ [8]. Hence, this is equivalent to generating the w_i tokens separately and choosing the maximum. To see it, first note that the probability distribution of the maximum of w_i tokens is the same as the one from $T^{(j)}$, where in this case it is $\Pr\left[t(z_i) \leq y\right] = y^{w_i}$.

Then, note that:

$$\Pr\left[t'(z_j) \leq y\right] = \Pr\left[\sqrt[w_i]{t(z_i)} \leq y\right] = \Pr\left[t(z_i) \leq y^{w_i}\right] = y^{w_i}$$

Hence both random variables have the same probability distributions. So by using the second approach, the asymptotic time cost of LotterySampling for the weighted versions doesn't increase.

Note that in the case of SpaceSaving, the time cost goes from $O(1)$ to $O(\log(m))$ since now the frequency of the elements doesn't increase by one and that makes its efficient data structure not suitable anymore.

3.5.4 LotterySampling adapted for the Heavy Hitters problem

In this thesis we have focused on the Top-k problem, specially when running the experiments. The proposed algorithm for LotterySampling can also be used for the Heavy Hitters problem, where we want to find all the elements with relative

frequency higher than ϕ.

However, we could use a simpler and more intelligent approach. We could forget the tickets and use a threshold t exactly like in SpaceSavingThreshold, but varying the threshold through the life of the stream, such that $t = \frac{\phi \cdot N}{\phi \cdot N + 1}$ (where N is the current length of the stream). In this version, t evolves exactly as the expected ticket of an element with frequency ϕ. Hence, the heavy hitters will have a higher expected ticket so we expect them to enter S.

4 Implementation

For this thesis, some of the most used algorithms for the Heavy Hitters and Top-k problems have been implemented, apart from the new ones proposed. The goal was to empirically compare the accuracy and efficiency of the algorithms through experiments using the defined metrics, with all of them sharing the same coding style and reusing as much as possible the underlying data structures.

The implementation of the algorithms has been done in C++. A test environment to run the algorithms with different streams has been done in Python. To visualize the results both Python and Matlab scripts can be used. The entire project can be found in
`https://github.com/GonMolon/Lottery_Sampling`.

4.1 Project structure

When building the project, an executable will be generated. To run it, some arguments need to be provided. These arguments will determine the used algorithm and its parameters, like the size of the sample S and other algorithm dependent parameters. Then, the stream is feed into the algorithm by writing into its standard input.

Every new-line-separated string represents an ID of some element, so each line will correspond to an item or apparition. The type of the IDs is parameterized and both strings and numerical types are supported.

The program will instantiate the chosen algorithm and it will be called with the corresponding ID for each item. All the algorithms need to implement an interface called GenericAlgorithmInterface. An abstract class called GenericAlgorithm is offered that already implements some of the methods from the interface, and offers some common functionalities between all the algorithms.

For example, GenericAlgorithm keeps a hash map to store and locate all the sampled elements in S. All the algorithm implementations inherit from Gener-

icAlgorithm and provide it the type of the sampled elements, which is a struct with all the necessary fields to keep the required information of each sampled element. Then, GenericAlgorithm will find the instance of an element given its ID using the hash map. If such instance doesn't exist, then it will call the abstract method "insert_element(Element)". In case this element was already being sampled, then the abstract method "update_element(Element)" is called.

The algorithm implementations basically just have to implement the two aforementioned abstract methods. The insert_element(Element) method will return a boolean indicating if the element needs to be kept in the sample by GenericAlgorithm. It may also remove another element if there is a replacement.

The algorithms also have to implement other methods to answer queries by writing on their standard output.

The implemented algorithms are:

- BasicLotterySampling
- ParallelLotterySampling
- SpaceSaving
- SpaceSavingThreshold (by generalizing SpaceSaving)
- LotterySampling
- Frequent
- CountSketch
- CountMin (by generalizing CountSketch)

4.2 Implementation details

The algorithms usually need to maintain the elements with some level of ordering. In fact, in some algorithms like LotterySampling more than one ordering is needed (order by frequency and by ticket). Hence, some data structures are needed to maintain such orders, each one with different properties and requirements. The list of implemented data structures is the following:

- BinaryHeap. It's an efficient custom implementation of a binary heap over a vector, such that the ordering key of the elements can be updated. It's useful for LotterySampling and others since the ticket of an element may increase while the element wasn't at the top of the heap. The operation of replacing an element is optimized. The elements can't be traversed in order by their key, since that operation is not required by a heap.

- SortedTree. It's a wrapper around the std::set. It's mainly used when we want: to traverse the elements in order by key, to insert an element with an arbitrary key, to arbitrarily modify the key of an arbitrary element and to delete an arbitrary element. It can also be used as a heap when it's required to traverse the elements in order. It is used by BasicLotterySampling to keep the elements sorted by frequency if $f_{init}(x) \neq 0$. It's also used by CountSketch and CountMinSketch.

- SortedList. It's an implementation of the data structure proposed in the SpaceSaving's paper. It consists in a list of buckets, where each bucket contains a list of elements. It's implemented as described in the original paper. It also has been described when describing SpaceSaving in this document. The key of an arbitrary element can increase by 1 unit each time. It's used by BasicLotterySampling when $f_{init}(x) = 0$ since it's also possible to remove an arbitrary element.

- SortedVector. It's a custom and improved implementation over a vector of the data structure proposed in SpaceSaving's paper. Instead of having a list of lists as suggested in the original paper, we just have a vector with all the elements sorted decreasingly by their keys. Hence, the elements with same key will be stored contiguously. We also have a list of buckets, but in this implementation a bucket doesn't contain a list inside. A bucket just keeps the index of the first element from the vector that is inside the bucket, and it also keeps its key (estimated frequency in our use cases). The list of buckets is also sorted by key, and each element also keeps a pointer to the bucket in which it is.

  Then, when an element increments its key, we get its bucket through its pointer. Then we place the element in front of all the elements that are in the same bucket. We do this by swapping it with the first element with the same key that appears first in the vector. We can find this element using the index that the bucket maintains. We update the index of the old bucket accordingly (by increasing it by one) or delete the bucket if it becomes empty. If the next bucket in the list of buckets already had the new key of the element, we just need to update the pointer of the element to point to this other bucket. If such bucket doesn't exist, we create it next to the old bucket in the list of buckets and we initialize its fields with the index and the new key of the element.

  This implementation is more efficient than the original implementation, both in memory footprint and execution time, since to have a vector instead of a list of lists is much more cache friendly and it requires less memory overhead. It's used by SpaceSaving, SpaceSavingThreshold and LotterySampling.

These data structures don't really keep the elements inside. They instead store pointers to the real elements which are located inside the hash map of Gener-

icAlgorithm. The struct that defines the elements needs to have fields called "locators" that are used to locate the elements inside every data structure in which they are. These locators are also modified by the data structures.

Also, as mentioned before, the tickets are represented with unsigned integers of 64 bits. The number of bits used affects the results because of resolution problems. Some functionalities about the tickets are shared between the algorithms so they are extracted in a TicketUtils class.

4.3 Test environment

The test environment is implemented in Python. There are experiments that measure the efficiency of the algorithms (time and memory consumption) and others that measure the accuracy.

To measure the efficiency the Valgrind profiler is used. When measuring both the memory and the time consumed by the algorithms, the cost of the IO operations is not accounted. That is accomplished by inspecting the output traces of Valgrind and ignoring everything that is outside the GenericAlgorithm class or subclasses.

Regarding the time cost that Valgrind reports: the value is not execution time, but number of instructions executed in a virtual machine, accounting cache misses, data reads, etc.

To measure the accuracy of the algorithms, it's necessary that the Python program keeps a counter for each element in the stream to exactly compute the heavy hitters and their exact frequencies. This is what all the proposed algorithms intend to avoid, since it requires a lot of memory. Hence, running experiments that measure the accuracy of the algorithms are very time consuming because the computer running the tests enters into swap memory. Also, the executions using Valgrind are very slow due to the virtualization. Some of the tests have been performed with EC2 instances in Amazon Web Services using hosts with more memory and more resources to address these problems.

The experiment results are saved in .csv files, and can be inspected and visualized with Matlab. A Matlab script is also provided to that end.

5 Experimentation

5.1 Streams

To experiment with the new and existing algorithms we have used streams generated from some Zipfian distribution. During the exploratory phase of the project we have also worked with streams generated from non-Zipfian distribu-

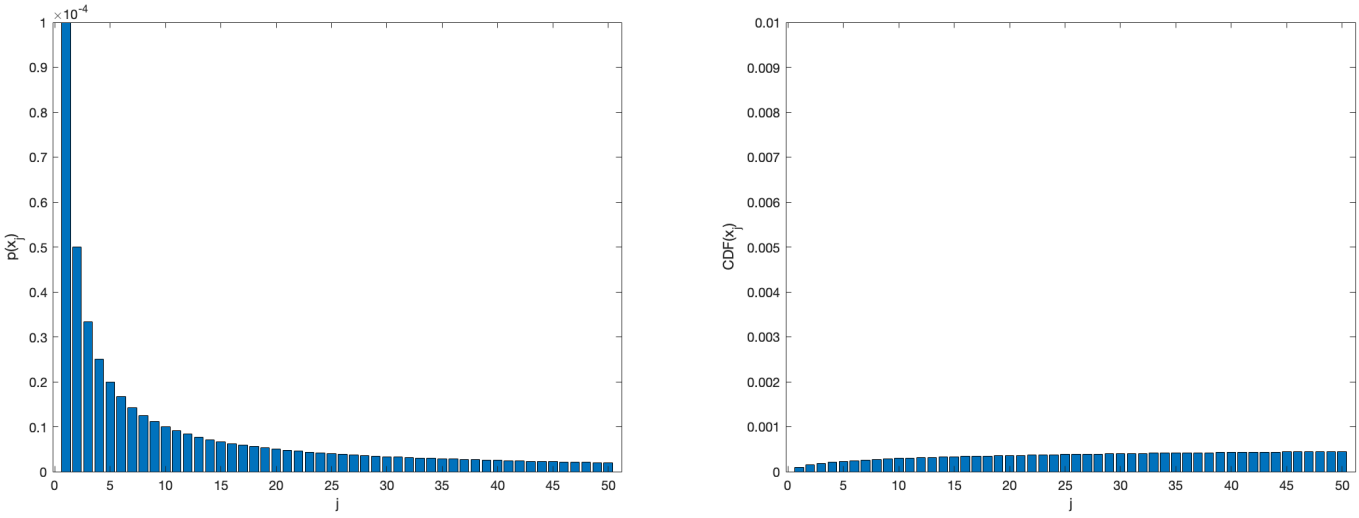
tions, but the results were very similar, and the Zipfian streams show well the most relevant features of the algorithms. Moreover, there is a well established "tradition" in the literature of using such Zipfian streams in experiments. Hence, the synthetic streams used in this documents are all Zipfian distributed streams.

We will refer to a stream that follows a Zipfian distribution with parameter α as Zipf-α-N, where N is the length of the stream. Hence, the theoretical relative frequency of the element x_j in a Zipf-α-N stream is:

$$p(x_j) = \frac{1/j^\alpha}{\sum_{v=1}^{\infty} 1/v^\alpha},$$

where $x_j = j \in \mathbb{N}$ and $\alpha > 1$.

In particular, we will use a few values for α. The probability density function and the cumulative probability distribution for $\alpha = 1.0001$ are plotted in Figure 3.



(a) PDF of Zipf-1.0001

(b) CDF of Zipf-1.0001

Figure 3: PDF and CDF of Zipf-1.0001 between x_1 and x_{50}

Note that we are only plotting the values for the top-50 heavy hitters, since the universe $\mathcal{U}$ is the entire set of naturals. Hence, the CDF doesn't arrive to 1 in its plot. Also note that for a fixed k, the frequencies of the top-k (heavy hitters) are much higher than the non-heavy hitters'. But the sum of the frequencies of the heavy hitters is still very low. This is why the Zipfian streams

are very interesting to experiment with.

We have also used a real stream from a big Internet company that we will call RealStream. The stream consists in a sequence of requests performed to one of the multiple clusters that are part of a distributed database. These requests were performed during 10 minutes on a day of high traffic, and only a 10% of the total number of requests were sampled. This stream has $N = 1.3 \cdot 10^9$ and its cardinality is $n = 271 \cdot 10^6$. The full streams occupies 100 GB of storage.

5.2 Experiments

Most of the experiments that have been realized during this thesis can be divided in three types:

1. For a single stream, some algorithms are executed once and some metrics are obtained by periodically measuring them. So in total there's only one execution per algorithm sharing the same stream, and by observing the obtained results we can appreciate how the metrics evolve as the stream is processed.

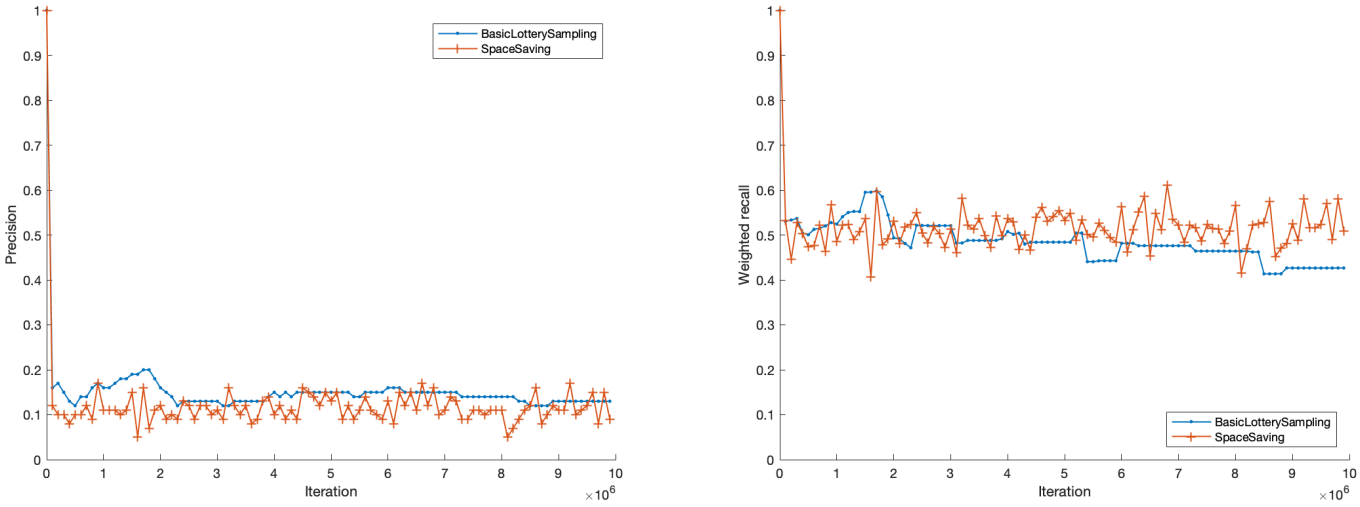
2. For a single stream, some algorithms are executed multiple times by modifying a common parameter between them, normally m (the size of the sample). At the end of each execution, the metrics are measured. Hence, this allows us to see how the modified parameter of the algorithms affect the metrics at the end of the stream.

3. A fixed set of algorithms with fixed parameters are executed multiple times, changing the stream in each execution. The metrics are measured at the end of each execution. In these experiments, the stream is normally synthetic and some parameter is changed. This is useful to see how the parameter of the stream affects the metrics. We will use this type of experiment varying the α of a Zipfian distribution.

During the exploration phase of the thesis we experimented with all the implemented algorithms. However, in the following results we will only include a subset of the algorithms, using the ones that performed best to compare our algorithms against them.

Experiment 1

The first experiment consists in comparing the accuracy of BasicLotterySampling and SpaceSaving with the stream Zipf-1.0001-10^7 in an experiment of type 1 using top-100 queries. Both algorithms share the same parameter $m = 200$. Their precision and weighted recall are plotted in Figure 4.



(a) Precision

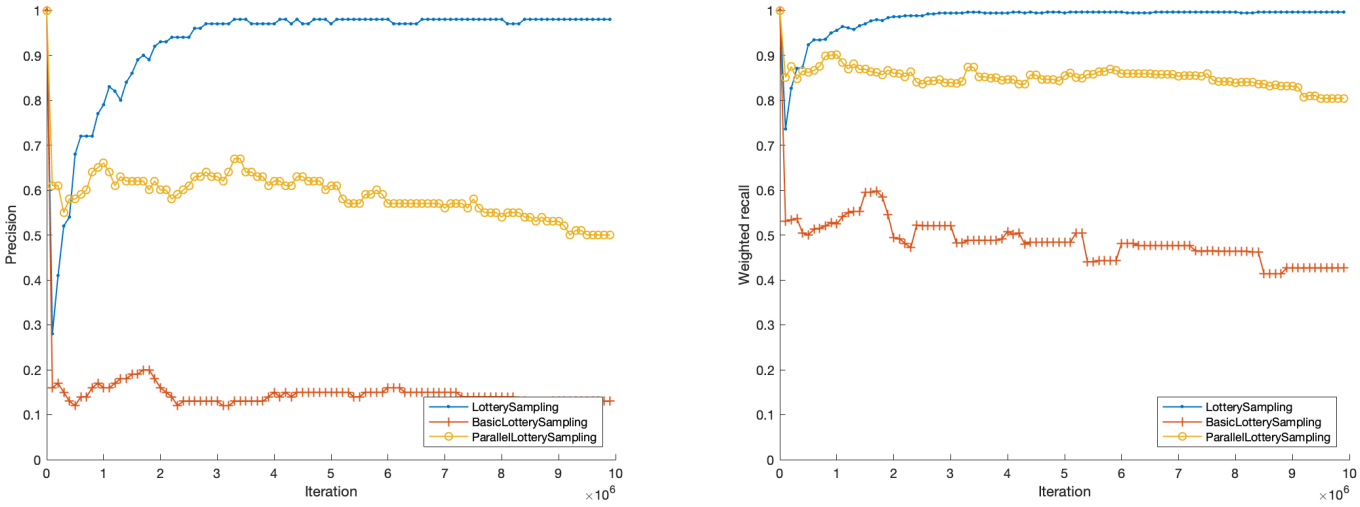
(b) Weighted recall

Figure 4: Accuracy top-100 in Zipf-1.0001-10^7

Note how BasicLotterySampling doesn't outperform SpaceSaving, having very similar values of precision and recall through all the stream. Also, note how their precision is very low, which means that they are returning just around 10% and 20% of the heavy hitters.

Experiment 2

In this experiment we repeat the experiment 1 but comparing BasicLotterySampling, ParallelLotterySampling and LotterySampling. All three algorithms share $m = 200$ and ParallelLotterySampling uses $h = 10$. The results are in Figure 5:



(a) Precision

(b) Weighted recall

Figure 5: Accuracy top-100 in Zipf-1.0001-10^7

Note how the results obtained by LotterySampling and ParallelLotterySampling are much better. In fact, LotterySampling almost achieves full precision and full weighted recall, meaning that it's practically solving exactly the Top-100 problem for this stream. And finally note that ParallelLotterySampling has 10 times more memory than LotterySampling.

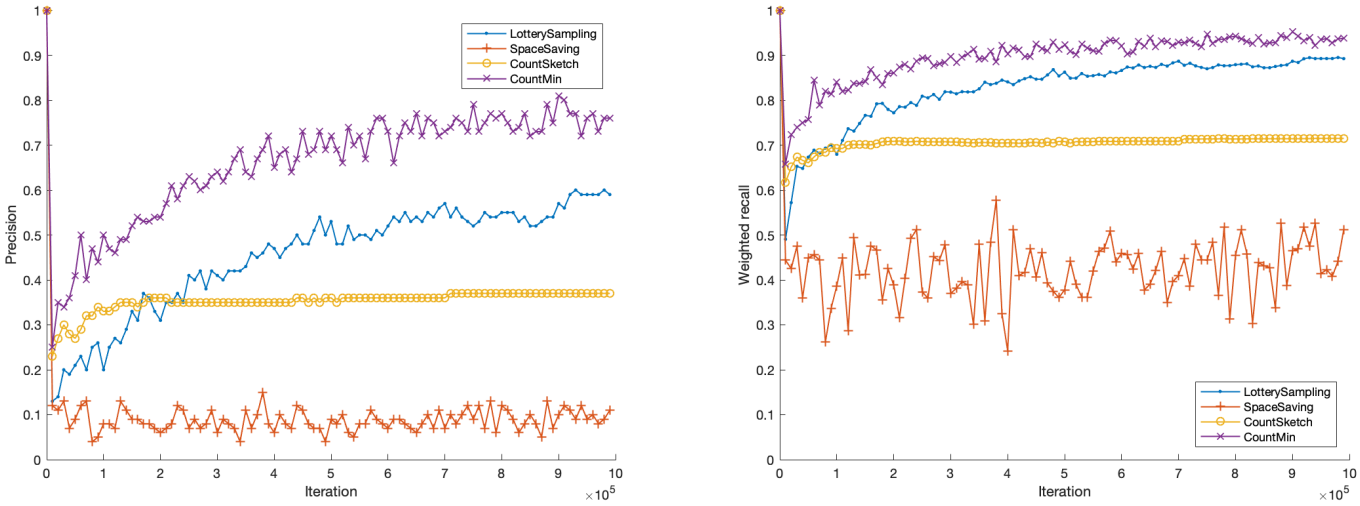
The accuracy of BasicLotterySampling and ParallelLotterySampling decreases slightly over time because the true heavy hitters are not always the same (the heavy hitters may change during the stream, if for example an element starts receiving many more hits that before), and the thresholds increase too fast (due to the effect of element alliance). Hence, their samples are not updated as they should because the threshold is already too high.

So this experiment tries to exemplify the superiority of LotterySampling against its simpler versions. From now on, we will not include BasicLotterySampling and ParallelLotterySampling in the rest of the experiments.

Experiment 3

In this experiment (of type 1) we compare LotterySampling, SpaceSaving, CountSketch and CountMin with a more "difficult" stream: Zipf-1.00001-10^6, which is less skewed and shorter than the previous ones. Hence, the distribution is rather "flat" with relatively small differences in frequency between the different elements.

All the algorithms share the parameter $m = 100$. SketchCount and CountMin also have parameters $h = 100$ and $q = 100$ which means that they will use a $h \cdot q$ matrix to store the shared counters. The results are shown in Figure 6:



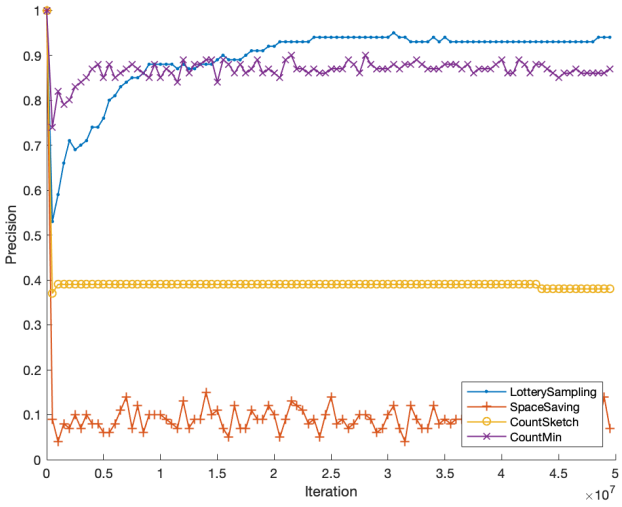
(a) Precision

(b) Weighted recall

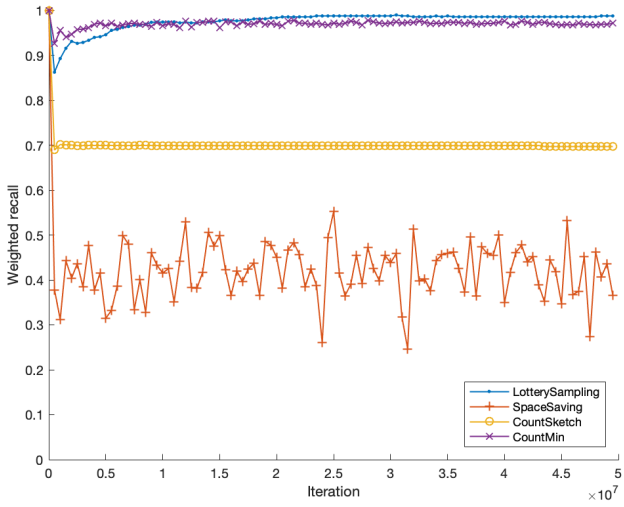
Figure 6: Accuracy top-100 in Zipf-1.00001-10^6

CountMin outperforms LotterySampling, both in precision and in weighted recall. However, note that CountMin and CountSketch use much more memory than LotterySampling and SpaceSaving (see Figure 8), since they keep a very big matrix apart from the sample S, so the comparison is not fair.

In Figure 7 we increase the length of the stream by 50, using the stream Zipf-1.00001-$50 \cdot 10^7$:



(a) Precision



(b) Weighted recall

Figure 7: Accuracy top-100 in Zipf-1.00001-$50 \cdot 10^7$

In this stream, which uses the same distribution as the previous one but it's longer, LotterySampling outperforms CountMin (with much less memory). What we try to exemplify here is the need of long streams that LotterySampling has when their distribution is difficult. So LotterySampling works very well even with difficult streams, but the more difficult they are, the longer they need to be.

This behaviour is mainly related to two things:

- $\text{Var}[T^{(j)}]$ will be larger in difficult streams because the elements will have less tokens, since their frequencies are smaller.
- The threshold of LotterySampling will also be smaller and it will increase more slowly for the same reason as before: The elements have less tokens so their expected tickets will also be lower.

When the streams are longer, the elements receive more tokens solving the aforementioned problem.

In appendix A we show the answer of every algorithm to a top-100 query at the end of the stream. It's useful as a visual inspection since the k heavy hitters are $1 \dots k$, so an algorithm that returns lower elements will have better accuracy.

The memory consumption of the algorithms in this experiment (with both streams) is shown in Figure 8:

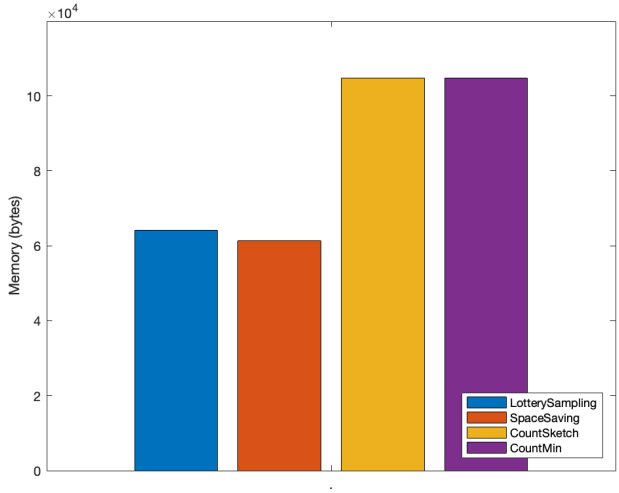
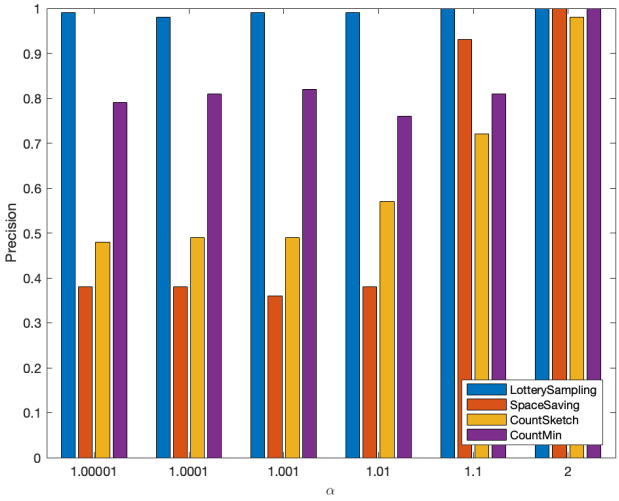


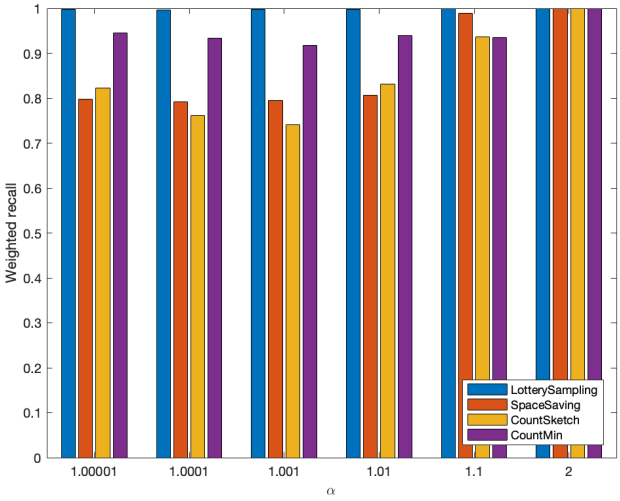
Figure 8: Memory usage

Experiment 4

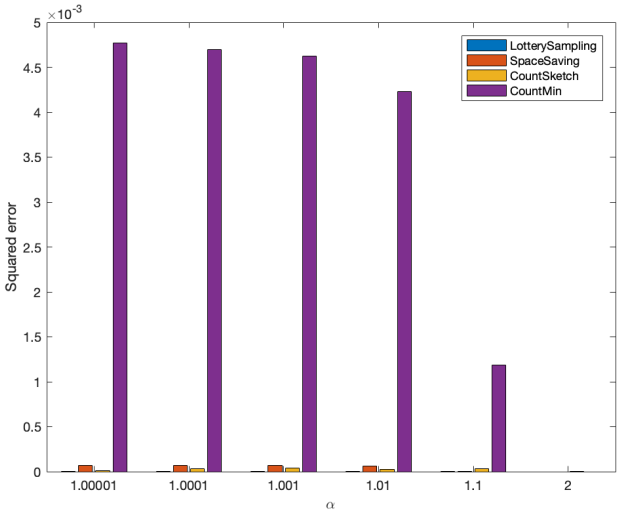
In this experiment (of type 3) we show how the accuracy of the algorithms increases when we increase the parameter α of Zipf-α-10^7. In all the executions, LotterySampling has $m = 200$, SpaceSaving has $m = 1000$, and CountSketch and CountMin have $m = 300$ and $h = q = 120$. This way, LotterySampling consumes half the memory than the other algorithms. The results are in Figure 9:



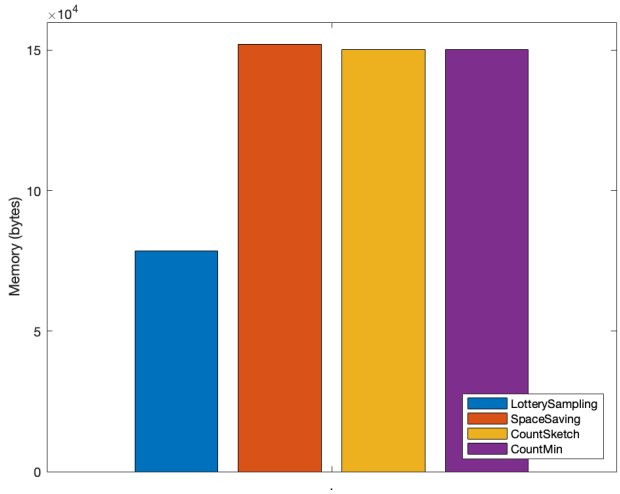
(a) Precision



(b) Weighted recall



(c) Squared error



(d) Memory usage

Figure 9: Accuracy top-100 in Zipf-α-10^7 varying α

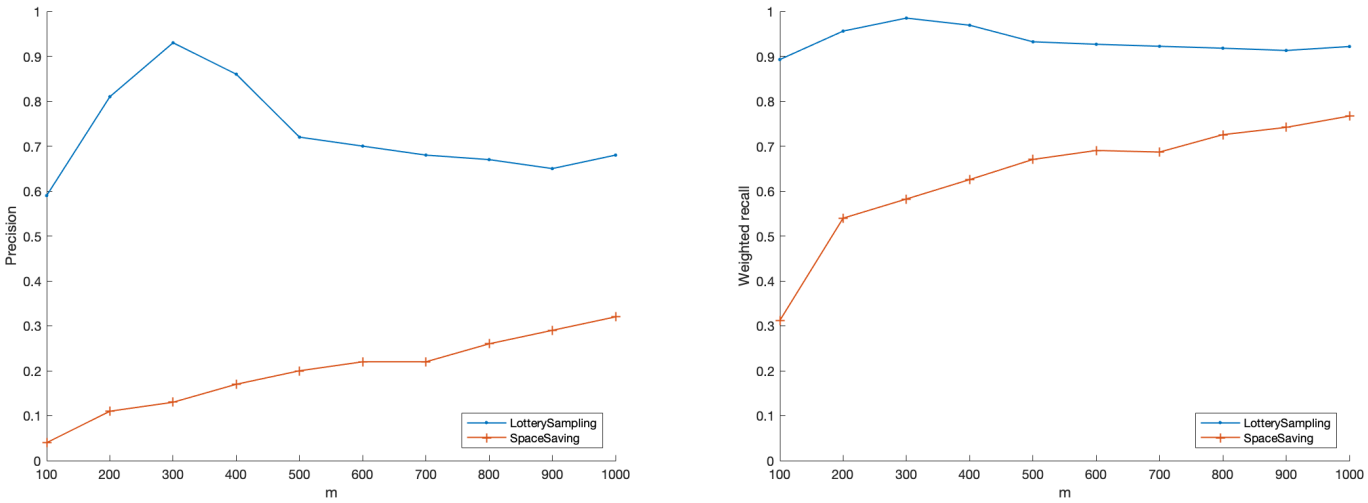
As expected, when increasing α the skewness increases, so the streams become easier and the algorithms perform better. Note how LotterySampling performs better than the rest, even with half the memory. Also note how CountMin, which seems to be the best existing algorithm in terms of precision and recall, has a very large squared error. This is due to the fact that it always overesti-

mates the frequencies of the elements, and the hash collisions will increase its estimations.

Experiment 5

In this experiment (of type 2) we aim to show how the size of the sample affects the accuracy of LotterySampling and SpaceSaving. Since LotterySampling almost always achieves full accuracy with long streams, in this experiment we will use a difficult and short stream for it to be interesting.

More specifically, we use a Zipf-1.00001-10^6, and we will vary the parameter m from 100 to 1000, always evaluating the accuracy with a top-100 query at the end of the stream. The results are shown in Figure 10.



(a) Precision

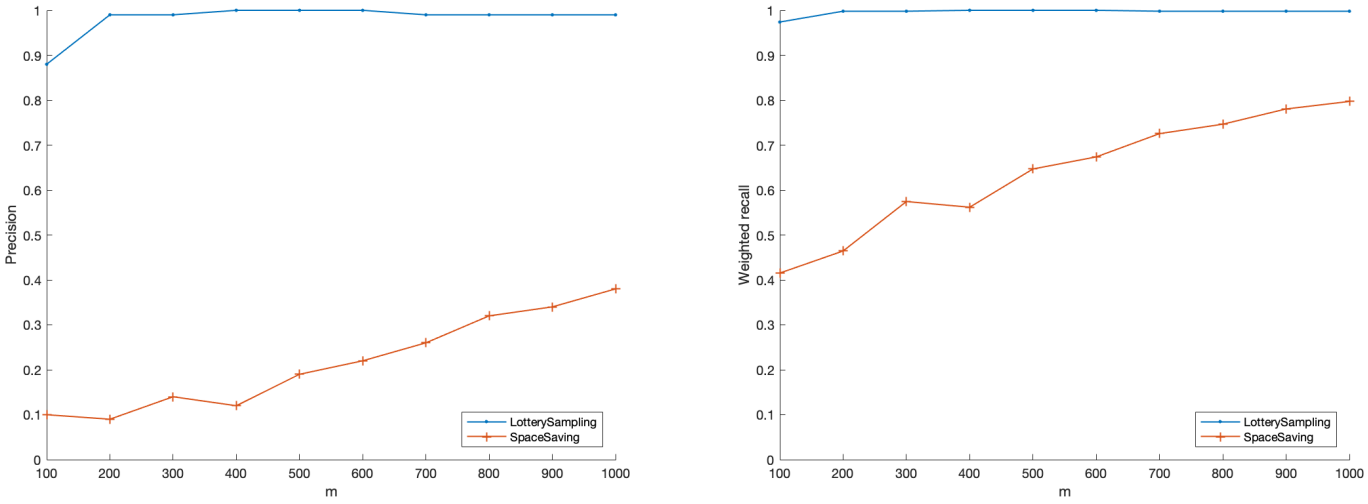
(b) Weighted recall

Figure 10: Accuracy top-100 in Zipf-1.00001-10^6 varying m

This is an interesting result related with the experiment 3. LotterySampling doesn't perform always better when increasing the size of the sample. This is a common behaviour of LotterySampling. The reason is that the threshold increases much slower when m is too big. This happens because the m^{th} heavy hitter has a lower frequency than the k^{th}, so its expected lottery ticket will increase more slowly. Hence, with such a low threshold, the frontier of LotterySampling moves too fast and the heavy hitters don't have time to run away from it.

Then, similarly as in experiment 3, if the length of the stream is long enough

for a given m, the accuracy will improve again. Hence, LotterySampling needs long enough streams when the streams are very difficult or its sample is very big. In Figure 11 we execute the experiment again with the stream Zipf-1.00001-10^7 which is the same but 10 times longer:



(a) Precision

(b) Weighted recall

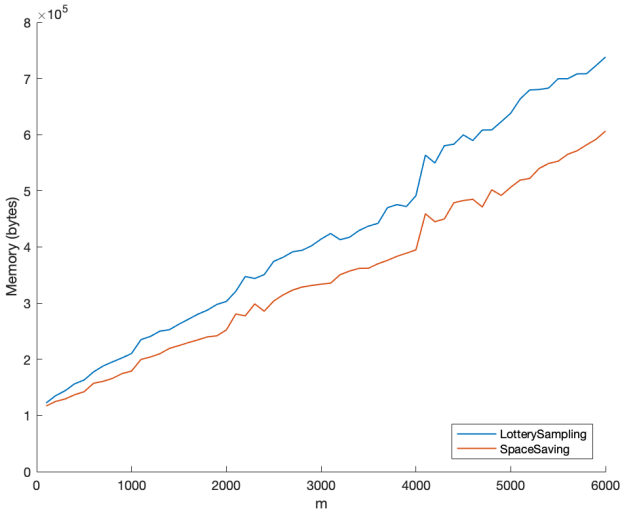
Figure 11: Accuracy top-100 in Zipf-1.00001-10^7 varying m

Now the counter-intuitive behaviour in which LotterySampling performed worse when increasing its sample doesn't occur anymore.

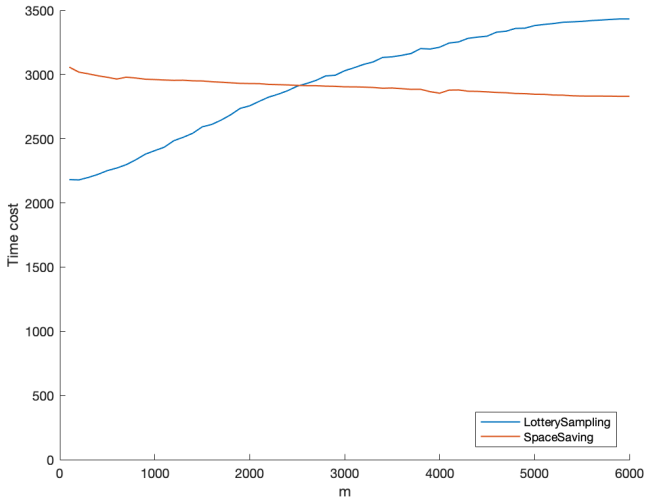
Experiment 6

In terms of efficiency, SpaceSaving is the best algorithm since it requires low memory (for a fixed m) and its asymptotic cost per element is $O(1)$. CountSketch and CountMin are the worst in this aspect, since they require quadratic amount of memory respect to h and q, and the asymptotic cost per element is $O(\log(m)+h)$. Note that the cost per element of LotterySampling is $O(\log(m))$.

In this experiment (of type 2) we show how the size of the sample affects the efficiency of LotterySampling and SpaceSaving. CountSketch and CountMin are excluded from this experiment since their efficiency is much worse. We use a Zipf-1.00001-10^7. The cost per item is the average. The results are plotted in Figure 12.



(a) Memory



(b) Time cost

Figure 12: Efficiency in Zipf-1.00001-10^7 varying m

The average cost per element of LotterySampling decreases through the life of the stream. This is due to its threshold never decreasing. So every time it will sample less items, having constant cost in such cases. Hence, if the stream were larger, the average cost per element of LotterySampling would be much smaller. (This explanation is not related with Figure 12 since there we are not modifying N, just m).

Also note how the time cost of SpaceSaving slightly decreases when increasing m. A possible explanation is that there will be a bit more updates instead of replacements since S is bigger. The hidden constants in an update are smaller than in a replacement.

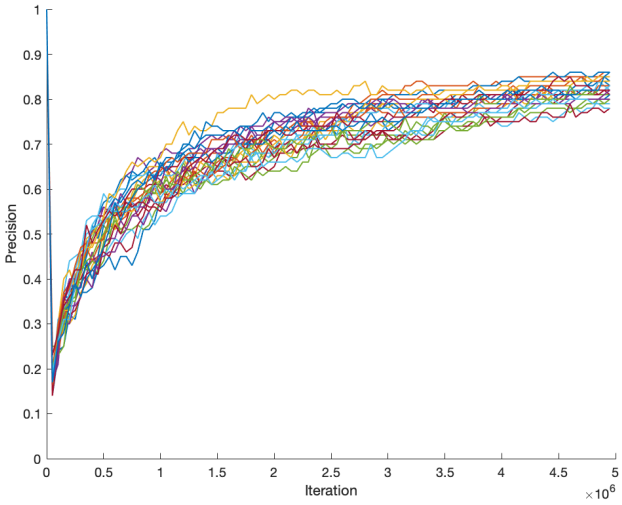
Experiment 7

We haven't given an expression for the variance of LotterySampling's accuracy (neither for other algorithms) since it's difficult and we considered it out of scope. However, we aim to estimate empirically the variance of LotterySampling by executing it 30 times varying its random seed, using always the same stream.

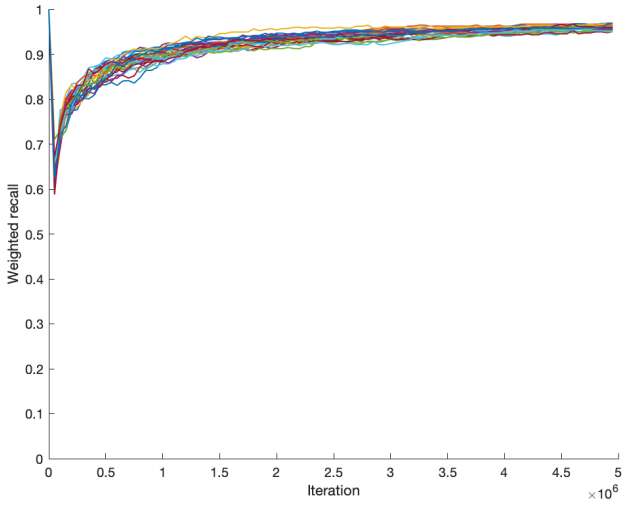
So for this experiment we will use the stream Zipf-1.0001-$5 \cdot 10^6$. Note that it's a short stream, and the variance would be smaller if the stream was longer.

In Subfigures 13a and 13b we show the evolution of precision and weighted recall through the stream for each of the 30 executions. In Subfigures 13c and

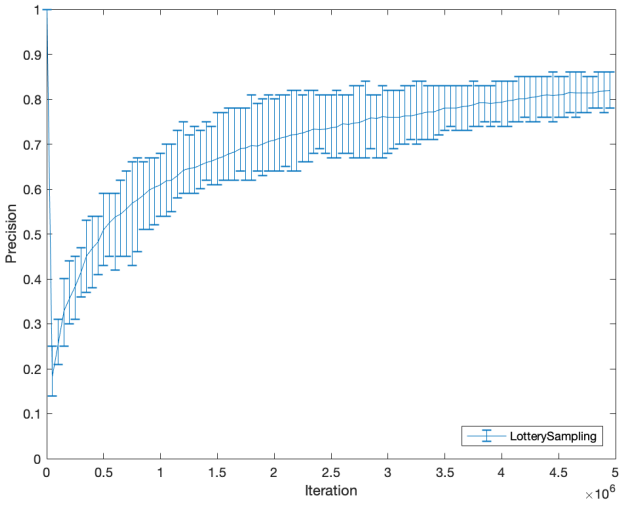
13d we show the averaged precision and averaged weighted recall through the stream, and the maximum and minimum values of the 30 executions.



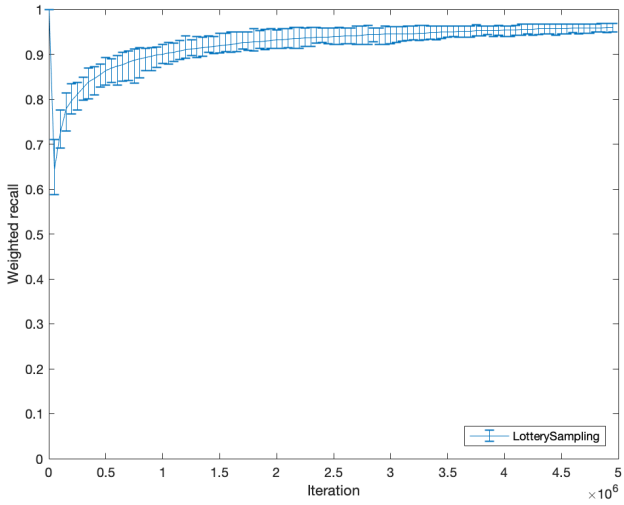
(a) Precision



(b) Weighted recall



(c) Precision



(d) Weighted recall

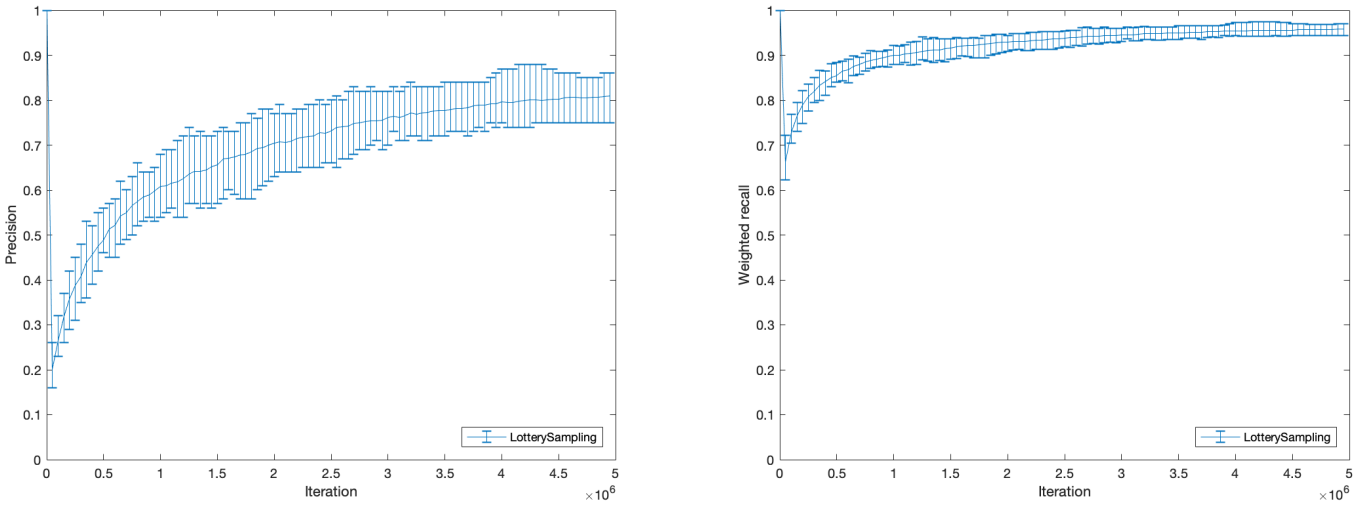
Figure 13: Accuracy variance top-100 in Zipf-1.0001-$5 \cdot 10^6$ changing the random seed

Notice that the variance of LotterySampling, specially in the weighted recall,

is very low.

Experiment 8

This experiment is very similar to the previous one, since we want to see how the variance of LotterySampling is when permuting the stream. Hence, we will also use Zipf-1.0001-$5 \cdot 10^6$ to run 30 times LotterySampling with it, but generating a random permutation of the stream for each execution. The results are shown in Figure 14.



(a) Precision

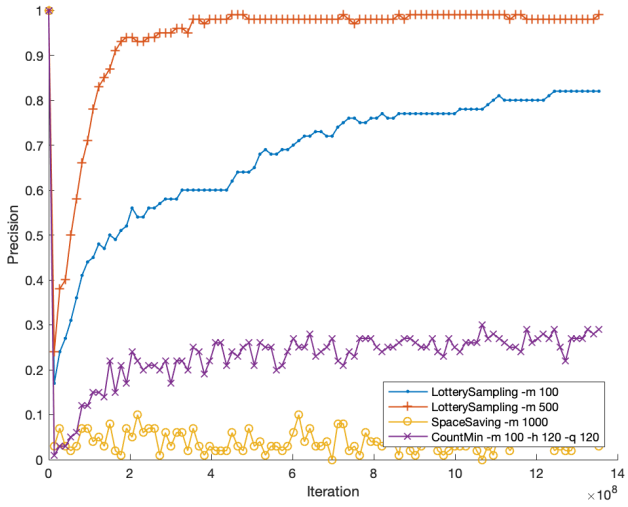
(b) Weighted recall

Figure 14: Accuracy variance top-100 in Zipf-1.0001-$5 \cdot 10^6$ changing the stream permutation

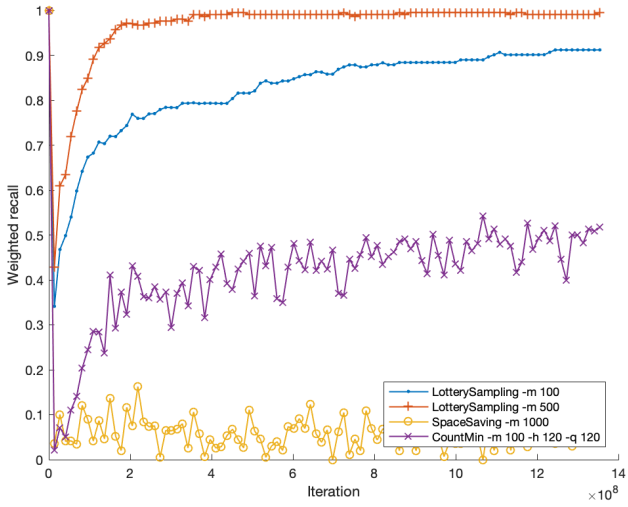
Note how the variance of LotterySampling is also low, very similar to the previous experiment, indicating that LotterySampling is resistant to different random permutations of the same stream. However, there exist permutations that harm a lot its accuracy. An example of a very harmful stream would be one with the elements appearing in decreasing order of frequency (so first all the heavy hitters and then the rest).

Experiment 9

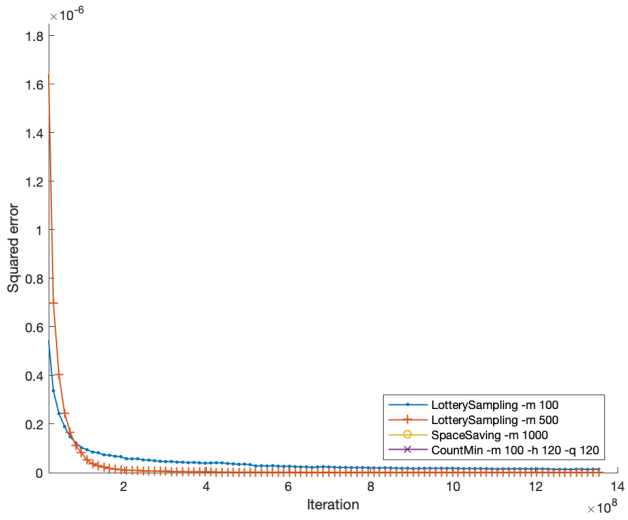
In this experiment (of type 1) we evaluate the accuracy of the algorithms with the real stream called RealStream described previously. We run 2 instances of LotterySampling with different parameter m. The results are plotted in Figure 15:



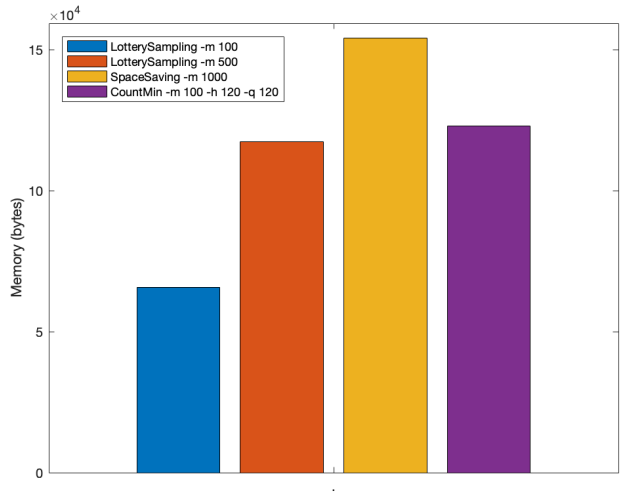
(a) Precision



(b) Weighted recall



(c) Squared error



(d) Memory usage

Figure 15: Accuracy top-100 in RealStream

6 Conclusions

LotterySampling is a novel algorithm based upon an original and innovative idea, and the experiments and its preliminary analysis indicate that it's far superior to previously existing algorithm in most practical settings, including very difficult streams.

We think that the given intuitions behind LotterySampling are convincing. However, much more efforts are required in giving formal proofs and theoretical guarantees of its performance.

There are also some variations (mentioned in this document) that are promising but they need to be tested and implemented more carefully, including the generalization for the weighted setups of the problems.

Hence, we don't consider that this work is finished and we will try to complete it in a near future.

Appendices

A Visual inspection of the algorithms' accuracy

The results of a Top-100 query at the end of a stream Zipf-1.00001-$50 \cdot 10^7$ are the following:

LotterySampling:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 59, 58, 60, 63, 62, 61, 64, 65, 67, 66, 68, 69, 71, 70, 73, 72, 74, 76, 77, 75, 78, 80, 79, 81, 84, 83, 86, 88, 91, 85, 92, 89, 93, 90, 95, 96, 101, 110, 113, 103, 42471302375, 27785328369207, 630932846465970

SpaceSaving:
1, 2, 406608324861036, 571345790840599872, 1792053523919, 40360142739, 112506041756, 11585509175, 885300673359616, 13770390610, 11142, 1916338767215777, 3181, 55481281460, 38827430309502, 648496, 432, 167783449753585376, 27775, 8074328, 1125303736512322, 145072778967054, 385414796126425600, 20356216809153, 478559793, 639341415366, 8650283365990666240, 1647617298061723136, 22681589409276404, 502817165, 1671261867591035, 210483659248207712, 6260691795, 18572494, 19718816374, 3787652552482222080, 5141148682431897600, 145947612739, 141052, 3393717, 2644849072780148224, 5717047601926281216, 3505082207190, 360956028, 636907448319784960, 60017807, 173276, 2660954, 461700860, 1242, 218576176488, 69, 197754970, 682857, 231612, 170765644519128032, 2714, 9230081060, 1375584820474042880, 781, 383844027, 5107852986673601, 2896286333170149, 17246625150675, 609587653490342, 23063174403431092, 204060, 11156745, 16338021345465308, 45257539992959, 17137676625147, 618026274174, 1172146731409, 1723604609, 4051204438, 146948750904, 6970, 129709027657, 8, 402925541, 316690, 49, 7621992477281129, 14813078892, 20560331, 798397436290895, 915103597069, 288591610041, 3024344681361, 1900634441, 524718935, 588496958006234, 1464332410, 221960461174, 16646142, 1980955507102574848, 677775, 5525799141262, 13947509, 719146236094

CountSketch:
1, 2, 3, 4, 5, 9, 11, 12, 13, 14, 15, 16, 17, 18, 22, 25, 26, 27, 29, 30, 31, 33, 34, 35, 36, 37, 38, 39, 41, 42, 44, 45, 50, 52, 56, 58, 62, 65, 126533705343901, 351767713733057, 879074770758411, 16475621214935404, 151282, 2512620024, 338222911753, 95124800877292, 32196446144768952, 24190367495936836, 2116798563, 395590558, 1001223944578818176, 6743688163, 16165977700754436, 9580489516304, 7461075738902, 1387788961621, 3256012180434895, 113852945060, 79971429004819, 35046848, 2305956, 2163356075851576, 382602275355, 892515918227, 26373382431875, 24988426, 376258839387, 23093621609528092, 93667146, 449922077746164608, 25265721459378, 82336306, 9581664239, 80589598, 304626731569216448, 96623, 656017908813, 268366119366301, 1303650026497355, 355378845045440832, 1873817819500824, 1040039, 633975963982394, 19022684024718792, 215379171, 1073206969626849280,

1361767665, 157580046, 135287, 14663764277, 7263412729, 928481, 13636609842090848, 231259172208101280, 6460698942130260, 38893398, 204998689909435, 4940165454648500224, 252113176622, 2057468849156536832

CountMin:
1, 6, 4, 5, 3, 12, 8, 11, 9, 15, 13, 2, 17, 26, 24, 29, 33, 28, 34, 43, 39, 21, 35, 32, 7, 38, 49, 16, 30, 10, 25, 40, 66, 22, 48, 46, 19, 67, 50, 44, 84, 80, 54, 59, 42, 87, 18, 56, 20, 53, 37, 71, 36, 14, 88, 90, 120, 69, 704125304292283776, 76, 61, 104, 62, 47, 83, 27, 51, 105, 64, 78, 52, 112, 65, 58, 70, 94, 73, 98, 74, 136, 91, 68, 57, 294066, 31, 79, 108, 23, 45, 106, 130, 55, 127, 492, 206, 41, 75, 63, 93, 155

B Scope and methodology

B.1 Scope

Initially, we planned to do this thesis about a new randomized algorithm invented by Prof. Conrado Martínez that we call BasicLotterySampling. This algorithm is inspired by the previous work of C. Martínez and his co-authors [1] in the related problem of cardinality estimation.

However, after the implementation of BasicLotterySampling we performed some tests comparing its efficiency and accuracy with the popular SpaceSaving algorithm. BasicLotterySampling was worse in both aspects. It required logarithmic asymptotic time respect to the size of the sample for each element in the stream, while SpaceSaving takes constant time. And the accuracy was notably worse.

Then, after the feedback gained with those tests, we designed a few more algorithms using the ideas from BasicLotterySampling and SpaceSaving. Combining them we obtained very promising results. On the preliminary tests, we observed a huge increase in both efficiency and accuracy with respect to the popular SpaceSaving.

After those results, we decided to focus on studying deeper the properties of our new algorithms and keep searching for variations and optimizations. These are the goals and results of this thesis:

1. Study previous work in the field. Understand the strengths and weaknesses of the existing algorithms. Understand the principles on which they are based.
2. Design new algorithms inspired by BasicLotterySampling to overcome its weaknesses, in order to obtain competitive algorithms that might outperform state-of-the-art algorithms.
3. Implement both our new and state-of-the-art algorithms. This implementation will be clean and efficient, written in C++ using OO programming paradigms. We will create a fully customized framework for them in which is easy to add new algorithms. All the implementations will share the same coding style so empirical comparisons are as fair as possible.
4. Analyze our new algorithms, both in terms of asymptotic time and memory.
5. Define metrics to measure the accuracy of these algorithms. It's important to find good and realistic metrics so the obtained results are reliable.
6. Create a test environment in which empirical results of time and memory are comparable between algorithms and executions. i.e. Use profilers with virtual machines to obtain these efficiency metrics.

7. Empirically compare our algorithms with the existing ones using the efficiency and accuracy metrics defined. These experiments will be carried out using a large variety of synthetic and real data streams.

Another of our goals was trying to carry out a mathematical analysis of the accuracy of our new algorithms. However, this has proven quite challenging, even for BasicLotterySampling, and therefore left out of the scope of this thesis. We will try, however, to provide strong arguments to explain why our main algorithm (LotterySampling) performs so extremely well.

B.2 Expected challenges

We believe that the domain we chose for this project is quite complex, specially because a lot of work has already been done and it's difficult to find new algorithms that improve the existing ones. However we have very promising results of the algorithms that we have discovered.

To think about optimal implementations and improvements to them will also be challenging, but doable. However, to give theoretical guarantees to our algorithms will likely be the most difficult goal, since our new algorithms are probabilistic, and proofs are specially difficult in this kind of algorithms.

Also, another challenging aspect is that there does not exist (to our knowledge) a common framework with "benchmarks" and commonly used data streams that helps us to compare algorithms in a standardized way. We will have to choose synthetic and real streams that may not reflect the complete truth. To solve this we will use a big variety of them with very diverse characteristics.

B.3 Methodology

Some Python scripts will be provided to run the experiments and to evaluate the algorithms in an automated way, offering detailed plots and stats based on the used metrics.

The results of the experiments will be used to guide the development and modifications of the algorithms, in a test driven development fashion.

The development will be done using Git and GitHub, so it can be easily tracked by the thesis director (C. Martínez). Also a ticketing system like Trello will be used to mark specific short-term goals. An agile approach to organize the development seems convenient, since a big part of the evolution of the work will be based on many new little ideas and changes supported by the experiment results. Since I have carried out this project while I was working in an internship in Luxembourg, I have kept very frequent communication with my advisor through Slack (messaging), Hangouts (for videoconferencing), and only occasionally through traditional e-mail or a shared Trello board.

C Project planning

The total time frame for this project is scheduled to be 7 months, starting on June 15^{th} 2018 and finishing on January 25^{th} 2019. Although some light planning and documentation has been realized before that date.

The entire project is divided in clear phases of: investigation, design, implementation, testing and theoretical analysis. However, due to the agile methodologies used, most of these phases are mixed and reviewed in later stages.

It's important to note that the author of this project starts an internship in September, so a big part of the work needs to be done during the Summer.

C.1 Tasks description and estimated time

1. Investigate about the Heavy Hitters and Top-k problems and understand existing algorithms to solve them. Read notes about BasicLotterySampling algorithm from C. Martínez. **50 hours**
2. Design a custom framework that allows:
   (a) Adding algorithms that share a common structure, sharing as much common code as possible (in C++).
   (b) Adding different metrics that evaluate the algorithms' performance.
   (c) Adding real and synthetic data streams.
   (d) Defining and running tests (in Python) using the implemented algorithms, metrics and data streams. Plotting the results visually afterwards.

   10 hours
3. Implement the previous mentioned framework. **20 hours**
4. Implement the algorithms SpaceSaving and BasicLotterySampling inside the framework. **20 hours**
5. Add a Zipfian data stream as an example to the framework and a subset of metrics to start testing. Iterativelly improve the framework and the two initial algorithms. **20 hours**
6. Choose and incorporate a reliable profiler (Valgrind) to the project so metrics about efficiency are accurate. This includes scripts that inspect the output of the profiler to find CPU and memory usage only used by the algorithms themselves, discarding other factors as IO or the OS interruptions. **10 hours**
7. Add stress tests with difficult data streams to test the weak characteristics of the algorithms. **20 hours**

8. Come up with ideas for new algorithms, implement them and identify which design decisions have more positive impact into the metrics using the outcome of the previous task. **70 hours**

9. Implement other already existing state-of-the-art algorithms. **30 hours**

10. From all the invented algorithms, choose the most promising one (or two). **10 hours**

11. Realize in-depth tests for the chosen final algorithm versions comparing them with the other existing algorithms. Document in detail all the tests, including their description and interpretation of the obtained results. **60 hours**

12. Try to reason theoretical guarantees and proofs about our chosen algorithms. **60 hours**

Most of these tasks have obvious dependencies from other tasks. So it's important to accomplish them in the stated order to avoid blocking dependencies. The total sum of estimated times is 390 hours.

Also, all of them just require the author of this project as human work force and a computer as a development station. Apart from that, some common software tools are going to be used, such as compilers (GCC), IDEs (CLion and SublimeText), profilers (Valgrind), etc.

C.2 Action plan

The action plan is represented using a Gantt chart in Figure 16. The assignment of days to tasks has been realized considering the total hours estimated for the task and the days in which it is going to be executed, since during the Summer there will be more working hours than during September for example (when the internship starts).

C.2.1 Changes in the planning

This project has a big part of invention and exploration of ideas for new algorithms, so it's easy that the initial plan gets modified depending on the discoveries and the experimentation.

In fact, as mentioned previously, there was a major change in the approach of the thesis during the first weeks of work, since we discovered the first algorithm with very good performance. After that point, we decided to keep exploring and the essence of the thesis changed.

However, there were no other major changes in the scope or planning of the project afterwards.

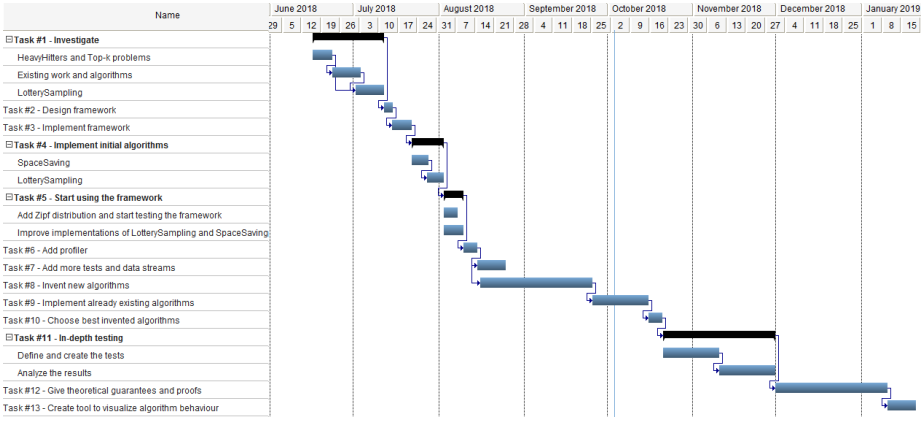


Figure 16: Gantt chart of the project. High resolution: https://goo.gl/GiQoJN

There was a minor delay in the initial planning because some tests were not possible to be executed in a local machine, since they required a lot of resources of memory and CPU. Hence, we had to move the development environment to an EC2 instance of Amazon Web Services in order to run the tests with the required resources cheaply.

There were no other deviations of the initial planning.

D Budget and sustainability

This work is a scientific research project, so it is not very straightforward to analyze it, regarding the economic impact, as a regular business activity. This is because it is practically impossible to estimate the costs in which other businesses incur using similar algorithms, and also to estimate the cost of the new ones. However, some analysis is possible and has positive results to all the implicated parties from an environmental, economic and social points of views. There hasn't been any changes from the initial prevision.

D.1 Environmental impact

The goal of this project is to find algorithms more efficient and accurate than the existing ones, which are extensively used in many areas as we already saw in previous chapters. This kind of algorithms are used without interruption on these industries, since they analyze very large or unbounded streams. So a small optimization in them will imply an important save of computational resources consumption. This will translate to less electricity usage, so a smaller environmental footprint. So in case they are adopted by the industry, their environmental impact will be positive, since they will decrease other business' footprint.

However, during the phase of the design and the experimentation of the new algorithms, some environmental resources will be used, and again, they are mainly electricity. Thanks to the nature of software this will be a fixed expense, and after this work is finished, the use of these algorithms will only save other resources, and it will easily compensate this initial consumption.

Anyways, a prediction of this initial environmental resources consumption is required. In order to obtain it, we will use the number of estimated required hours to develop this project calculated on previous chapters. Apart from those human-working hours, we also have to consider all the time required for running the tests and experiments, which is notably a lot. The sum of those hours, with the estimated hourly consumption of electricity by a personal computer will be the prediction of used electricity.

It's difficult to estimate the required amount of hours to run the tests, but I expect to run a test every night of the second half of the development of this project. Following the Gantt chart, this means that I will start with this time consuming tests in September-October, during 105 days. These tests consist on streams of one hundred million elements approximately, and they usually last 7 hours.

$$\text{Testing hours} = \frac{\text{Estimated project days}}{2} \frac{7 \text{ hours}}{\text{night}} = 367 \text{ hours}$$

$$\text{Computing hours} = \text{Working hours} + \text{Testing hours} = 390 + 367 = 757 \text{ hours}$$

$$\begin{aligned}\text{Electricity} &= \text{Computing hours} \cdot \text{Mean power consumption} \\ &= 757 \text{ hours} \cdot 0.1 \text{ kW} = 75.7 \text{ kWh}\end{aligned}$$

Regarding physical resources, I will use my personal computer without needing any additional hardware. So in this sense, there will be no footprint coming from buying new hardware.

D.2 Economic impact

As discussed previously, it's difficult to elaborate a detailed analysis of the economic impact of the project on its useful life. But anyways, we can detail a budget (based on the estimated costs) to realize this investigation project.

The main cost of this project will come from the human working hours, which is a direct cost. More specifically, from the student and professor authors of this thesis. The amount of hours spent by the student will be the ones detailed on the previous chapter: 390. The hours spent by the professor will be estimated as 1/8 of the student hours, ie approximately 49 hours, since he will have to actively review the work done by the student, discuss new ideas, etc.

Then, considering an hourly salary of 25 euros for the student, and 50 for the professor, we get a direct cost of:

$$\text{Direct cost} = 390 \text{ hours} \cdot \frac{25 \text{ euros}}{\text{hour}} + 49 \text{ hours} \cdot \frac{50 \text{ euros}}{\text{hour}} = 12200 \text{ euros}$$

The only indirect economic expense will be related to the consumption of electricity. This, at the same time, will depend on the hours required to develop the algorithms and to run the experiments. From the environmental analysis, we know the total expected consumption of electricity, and we will use that metric with the mean hourly cost of electricity to estimate the economic impact.

$$\text{Indirect cost} = 75.7 \text{ kWh} \cdot \frac{0.07 \text{ euros}}{\text{kWh}} = 5.3 \text{ euros}$$

Then, we can calculate the total costs as:

$$\text{Total estimated cost} = 12200 \text{ euros} + 5.3 \text{ euros} = 12205.30 \text{ euros}$$

And adding a 8% of that quantity as contingency costs, we get as final budget:

$$\text{Total budget} = \text{Total estimated cost} \cdot 1.08 = 13185.54 \text{ euros}$$

D.3 Social impact

Depending on the use of the algorithms proposed in this project, the social impact will be different. The algorithms solve a theoretical problem present in data streams, and as we saw on previous chapters, this problem can be found in very different real situations. These algorithms aim to solve more accurately the problem, so the entities using them in a future will be able to get better results. That can translate to detecting better DDoS attacks, or decreasing the latency of Internet services. So in theory this project will have a positive social impact (although very indirectly) to possibly many people.

From a personal perspective, the authors of this project will expand their knowledge and skills. If the proposed algorithms turn out to be successful, they may receive recognition from other researchers.

References

[1] A. Helmi, J. Lumbroso, C. Martínez, and A. Viola. "Data Streams as Random Permutations: the Distinct Element Problem". In N. Broutin and L. Devrye, editors. *Proc. 23rd Int. Conf. on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms (AofA)*. Discrete Mathematics & Theoretical Computer Science (Proceedings), pages 323–338, 2012.
https://hal.inria.fr/hal-01197221/document

[2] A. Metwally, D. Agrawal, and A. E. Abbadi. "Efficient Computation of Frequent and Top-k Elements in Data Streams". In International Conference on Database Theory, pages 398–412, 2005.
https://pdfs.semanticscholar.org/72f1/5aba2e67b1cc9cd1fb12c99e101c4c1aae4b.pdf

[3] E. Demaine, A. López-Ortiz, and J. Munro. "Frequency Estimation of Internet Packet Streams with Limited Space". In Proceedings of the 10th ESA Annual European Symposium on Algorithms, pages 348–360, 2002.
https://pdfs.semanticscholar.org/ab52/1e91a2bce4fd13326cba9a765b479feefd61.pdf

[4] M. Charikar, K. Chen, and M. Farach-Colton. "Finding Frequent Items in Data Streams". In Proceedings of the 29th ICALP International Colloquium on Automata, Languages and Programming, pages 693–703, 2002.
https://www.cs.rutgers.edu/∼farach/pubs/FrequentStream.pdf

[5] G. Cormode and S. Muthukrishnan. "An improved data stream summary: The count-min sketch and its applications". In Journal of Algorithms, 55(1):58–75, 2005.
http://dimacs.rutgers.edu/ graham/pubs/papers/cm-full.pdf

[6] G. Manku and R. Motwani. "Approximate Frequency Counts over Data Streams". In Proceedings of the 28th ACM VLDB International Conference on Very Large Data Bases, pages 346–357, 2002.
http://www.vldb.org/conf/2002/S10P03.pdf

[7] M. Fischer and S. Salzberg. "Finding a Majority Among N Votes: Solution to Problem 81-5". In Journal of Algorithms, 3:376–379, 1982.
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.3439

[8] L. Devroye. "Non-Uniform Random Variate Generation". In Springer-Verlag, New York, 1986.
http://www.nrbook.com/devroye/